Realizing the Vision of Zero Software Defects

Systems & Software Technology Conference Tutorial

Jay Abraham
jay.abraham@mathworks.com

May 16th 2011

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **16 MAY 2011**	2. REPORT TYPE	3. DATES COVERED **00-00-2011 to 00-00-2011**

4. TITLE AND SUBTITLE **Realizing the Vision of Zero Software Defects**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **MathWorks,3 Apple Hill Drive,Natick,MA,01760**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
Presented at the 23rd Systems and Software Technology Conference (SSTC), 16-19 May 2011, Salt Lake City, UT. Sponsored in part by the USAF. U.S. Government or Federal Rights License

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**103**	

Tutorial Agenda

- Complexity of Systems
  - Failures and their cause
- Implementation and Verification
  - Developing robust systems
- Model and Code Verification
  - Addressing design and code errors
- Practical Considerations
  - Implementing and verifying complex systems
- Additional Techniques for Improving Software Quality
  - Addressing standards and other considerations

Complexity of Systems

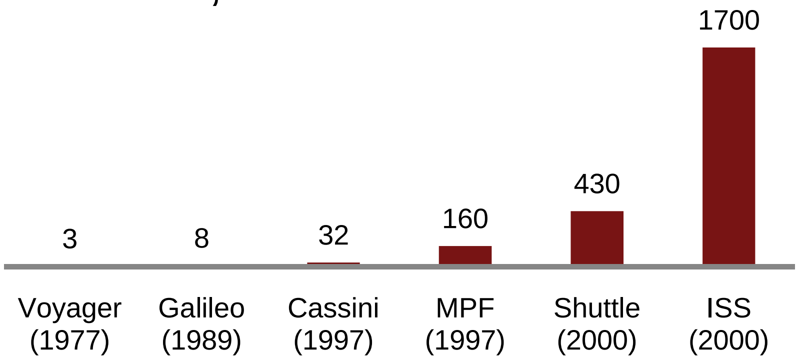
Failures and their cause

Complexity of Systems

- Modern automotive powertrain
  - 500 to 1,000 thousands lines of code (KLOC)

- Boeing 787 flight control system
  - 6,500 KLOC

- Software in spacecraft*
  - 3 to 1,700 KLOC



*Automated Software Verification & Validation: An emerging approach for ground operations
Bell and Brat, NASA

Complex Systems Fail

- Ariane-5, expendable launch system
  - Overflow error
  - Resulted in destruction of the launch vehicle

- USS Yorktown, Ticonderoga class ship
  - Divide by zero error
  - Caused ship’s propulsion system to fail

- Therac-25, radiation therapy machine
  - Race condition and overflow error
  - Casualties due to overdosing of patients

Cost of Failure – Aerospace Examples*

System	Cost	Reason
Ariane 5 (1996)	$594M	Overflow software error
Delta III (1998)	$336M	SW did not account for normal roll oscillation
Titan IV B (1999)	$1.5B	Wrong decimal point in SW (const -0.19.. vs. -1.99..)
Mars Climate Orbiter (1999)	$524M	Wrong units
Zenit 3SL (2000)	$367M	Premature 2nd stage shutdown
Messenger (2004)	$24M	SW test related delays resulting in data loss

*Automated Software Verification & Validation: An emerging approach for ground operations
Bell and Brat, NASA

Why Do Complex Systems Fail?*

- Insufficient specification
- Design errors
- Software coding errors
- Mechanical failure
- Deliberate interference
- Human errors

*Issues in Safety Assurance
Martyn Thomas, SafeComp 2003

Scope of Tutorial

- Insufficient specification
- Design errors
- Software coding errors
- Mechanical failure
- Deliberate interference
- Human errors

⬅ Embedded Software

Design Errors

- Poorly designed software
  - That may or may not adhere to specifications
- Avoiding design errors
  - Not easy, issues may not be detected
  - With non-exhaustive testing or simulation methods
- Effects include
  - Software crashes
  - Unexpected software behavior

Design Error Examples

- Dead logic
- Unreachable states
- Deadlock conditions
- Non-deterministic behavior
- Exception conditions

- Overflow
- Divide by zero
- And lots more …

Software Code Errors

- Coding defects
  - Resulting in run-time errors
- What are run-time errors
  - Also known as “latent faults”
  - Rarely manifest and are infrequent
- Effects include
  - Software crashes
  - Unexpected software behavior

Run-Time Error Examples

- Non-initialized data
- Out of bound array access
- Null pointer dereference
- Incorrect computation
- Concurrent access to shared data

- Illegal type conversion
- Dead code
- Overflows
- Non-terminating loops
- And lots more …

The Vision of Zero Defect Software

- Is it possible?
- Yes, but with some caveats

- Is it applicable to all types of software?
- No, and that’s OK

- So when does it make sense to invest time, energy, and effort to create zero defect s/w …

Constraining the Problem

- When does software quality truly matter
  - Human lives at risk
  - Missions that cannot fail
  - Business operations that cannot suffer downtime

- Computer devices
  - High integrity embedded systems
  - Examples: flight control, braking systems, remote cellular base stations, …

Introduction to High Integrity Embedded Systems

- General embedded systems
  - Software world-wide increasing 10% to 20% per year
  - Embedded microprocessors account >98%
- High integrity systems found in
  - Aircraft, automobiles, medical devices
  - Safety and reliability are paramount
- Software algorithms contain
  - Complex controls algorithms
  - Computations in fixed point and floating point
  - Logic, state based machine algorithms
  - Multi-threaded code execution

Challenges in High Integrity*

- Strong correlation between application size and the total number of defects
  - Estimated 30 defects per 1000 lines of code
  - 20% will be severe
  - Defects must be found and removed

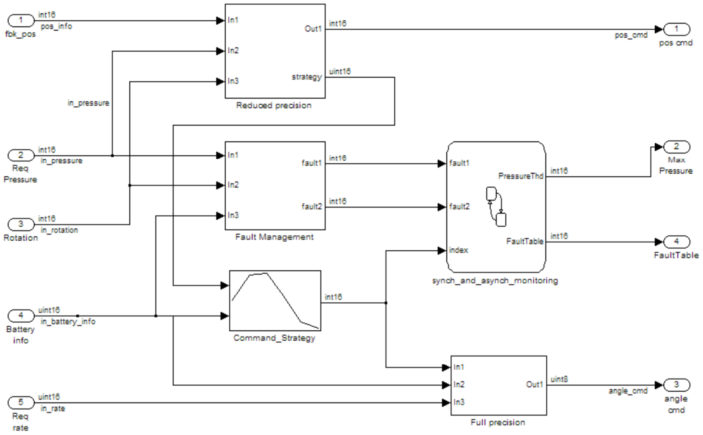
- Time and resources allocated to finding and fixing software defects
  - Most expensive aspect of software development

* Embedded software: facts, figures, and future
Ebert And Jones, IEEE Computer 2009

Implementation and Verification of Complex Systems

Implementing and Verifying Complex Embedded Software Systems

Software for an Engine Controller

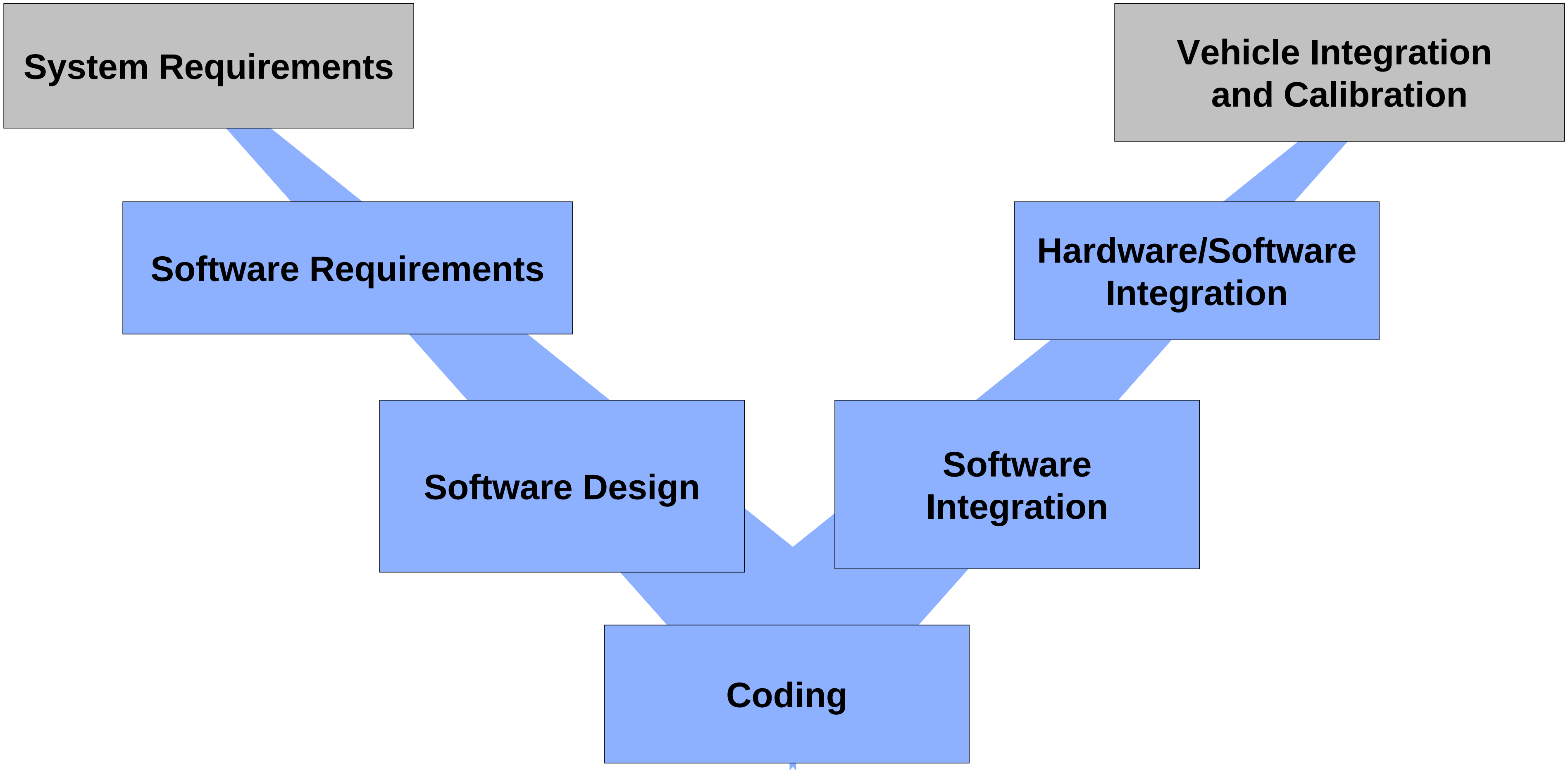


Complex Algorithm

Embedded Controller

Aircraft Engine

Design Implementation and Verification



System Requirements

Software Requirements

Software Design

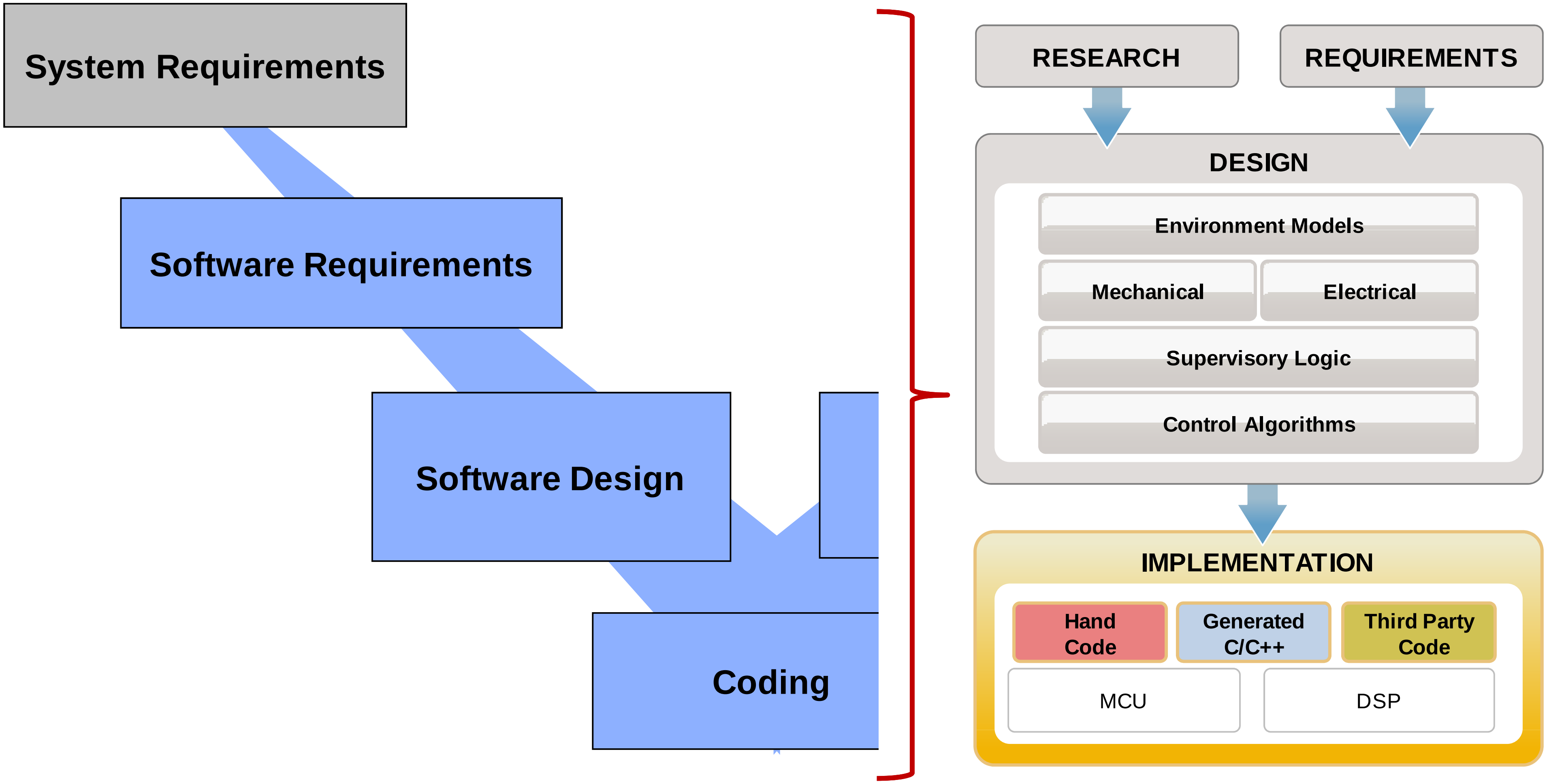
Coding

Software Integration

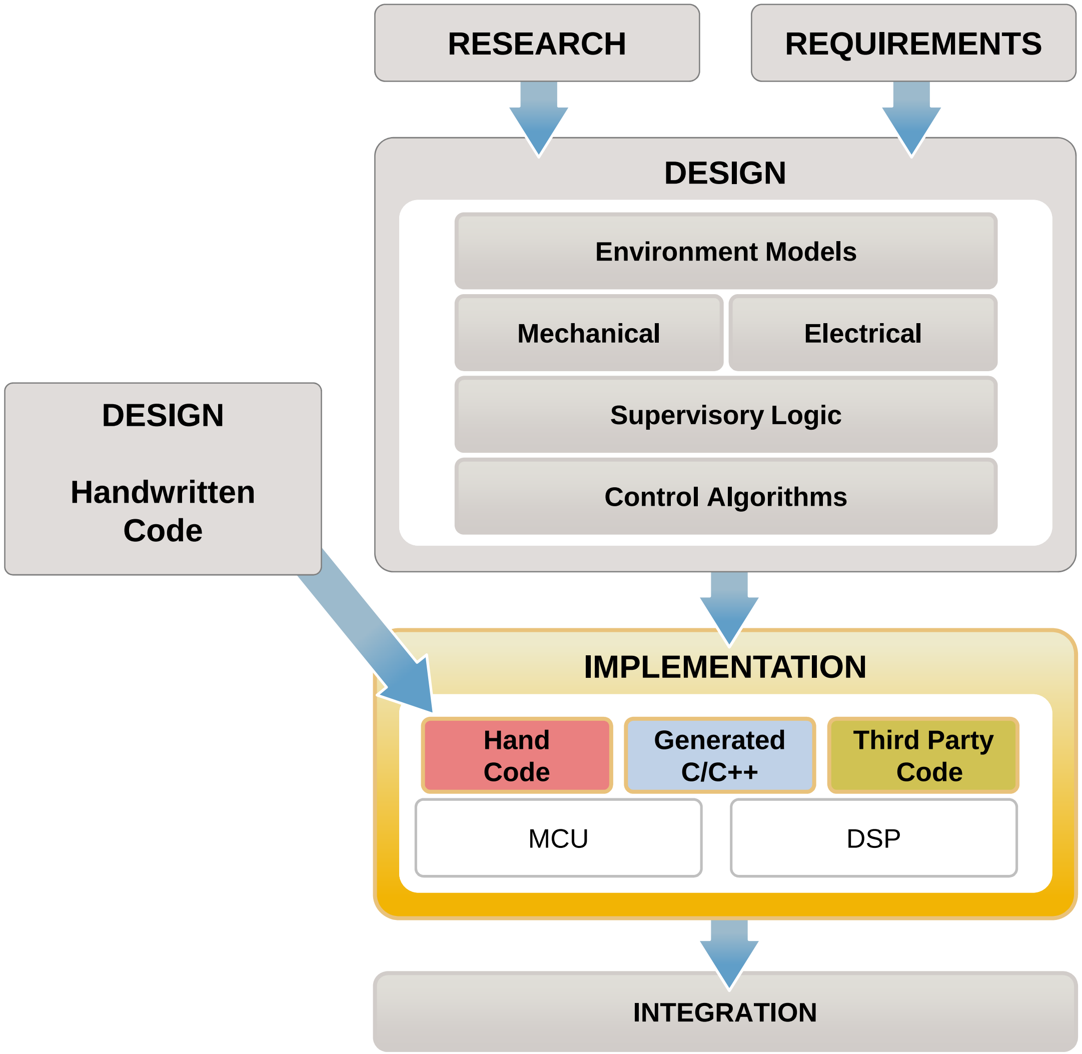
Hardware/Software Integration

Vehicle Integration and Calibration

Design Implementation



Design Implementation with Model Based Design (MBD)



RESEARCH

REQUIREMENTS

DESIGN

Environment Models

Mechanical

Electrical

Supervisory Logic

Control Algorithms

DESIGN

Handwritten Code

IMPLEMENTATION

Hand Code

Generated C/C++

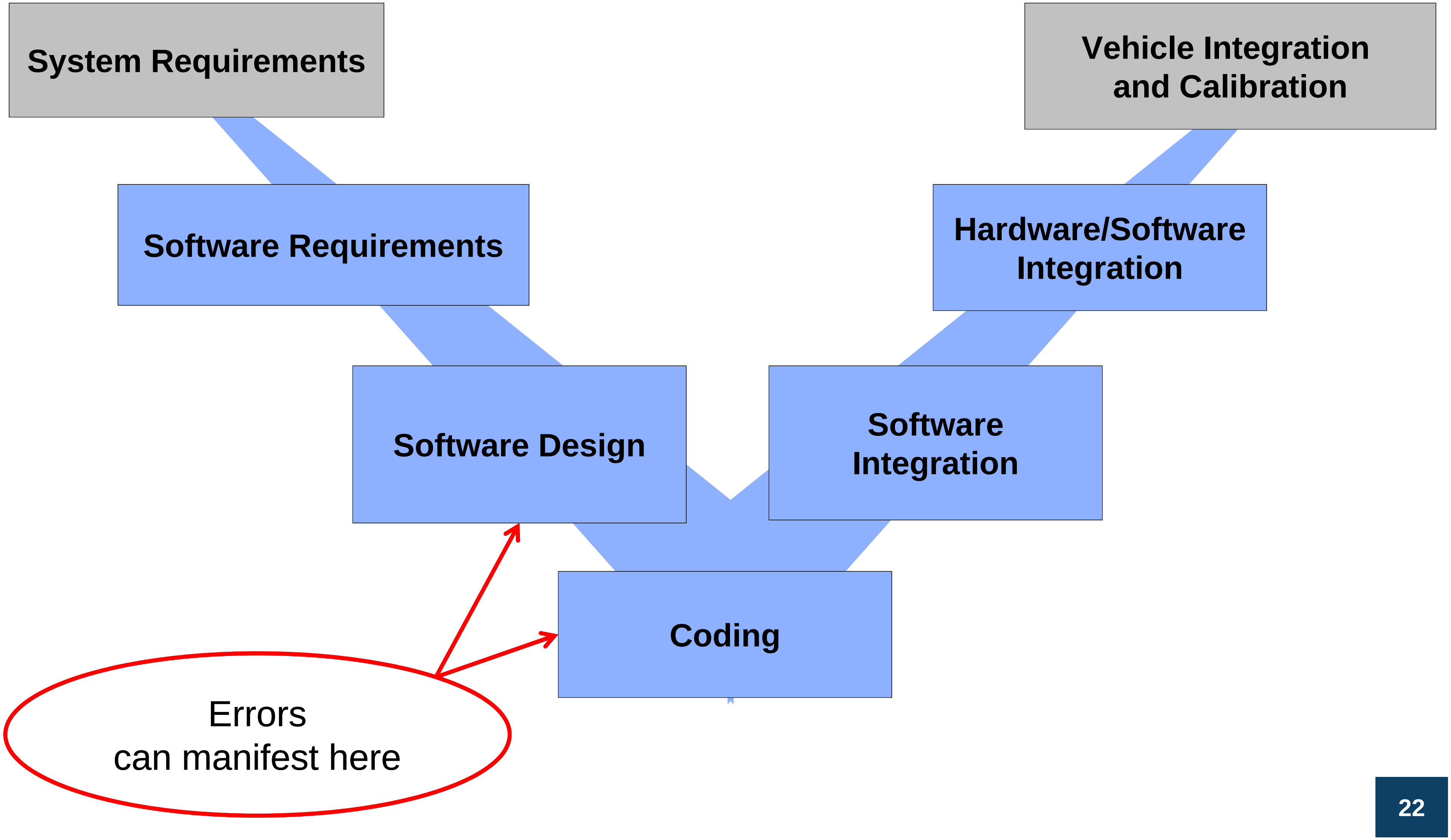
Third Party Code

MCU

DSP

INTEGRATION

Design & Code Error Manifestation



System Requirements

Software Requirements

Software Design

Coding

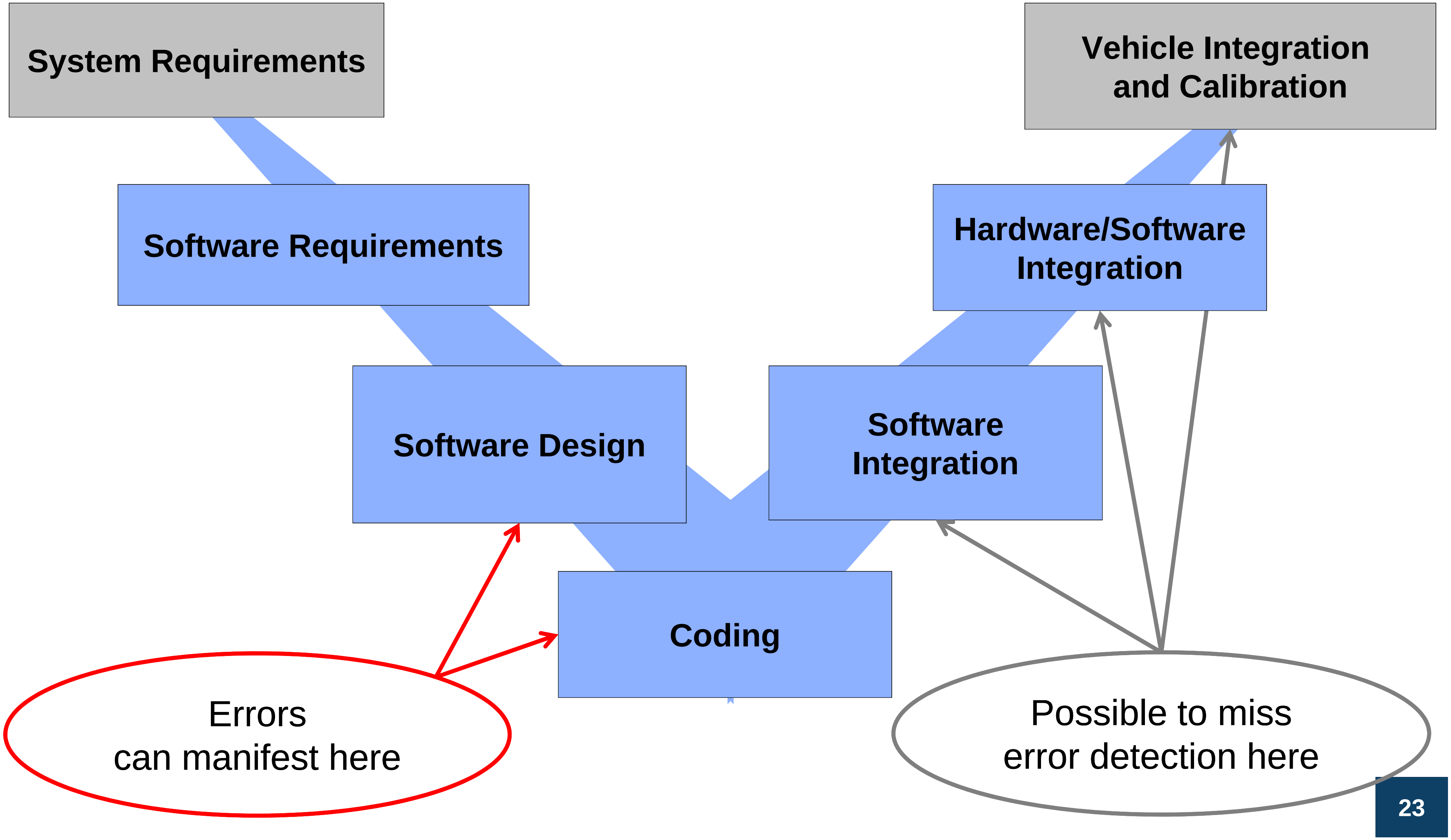
Software Integration

Hardware/Software Integration

Vehicle Integration and Calibration

Errors can manifest here

Design & Code Error Detection



System Requirements

Software Requirements

Software Design

Coding

Software Integration

Hardware/Software Integration

Vehicle Integration and Calibration

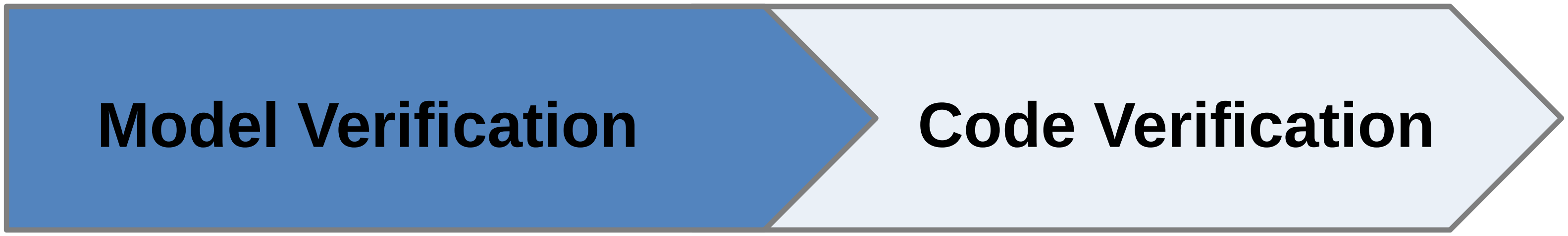
Errors can manifest here

Possible to miss error detection here

Model and Code Verification

Addressing design and code errors

Solving the Problem with Model and Code Verification

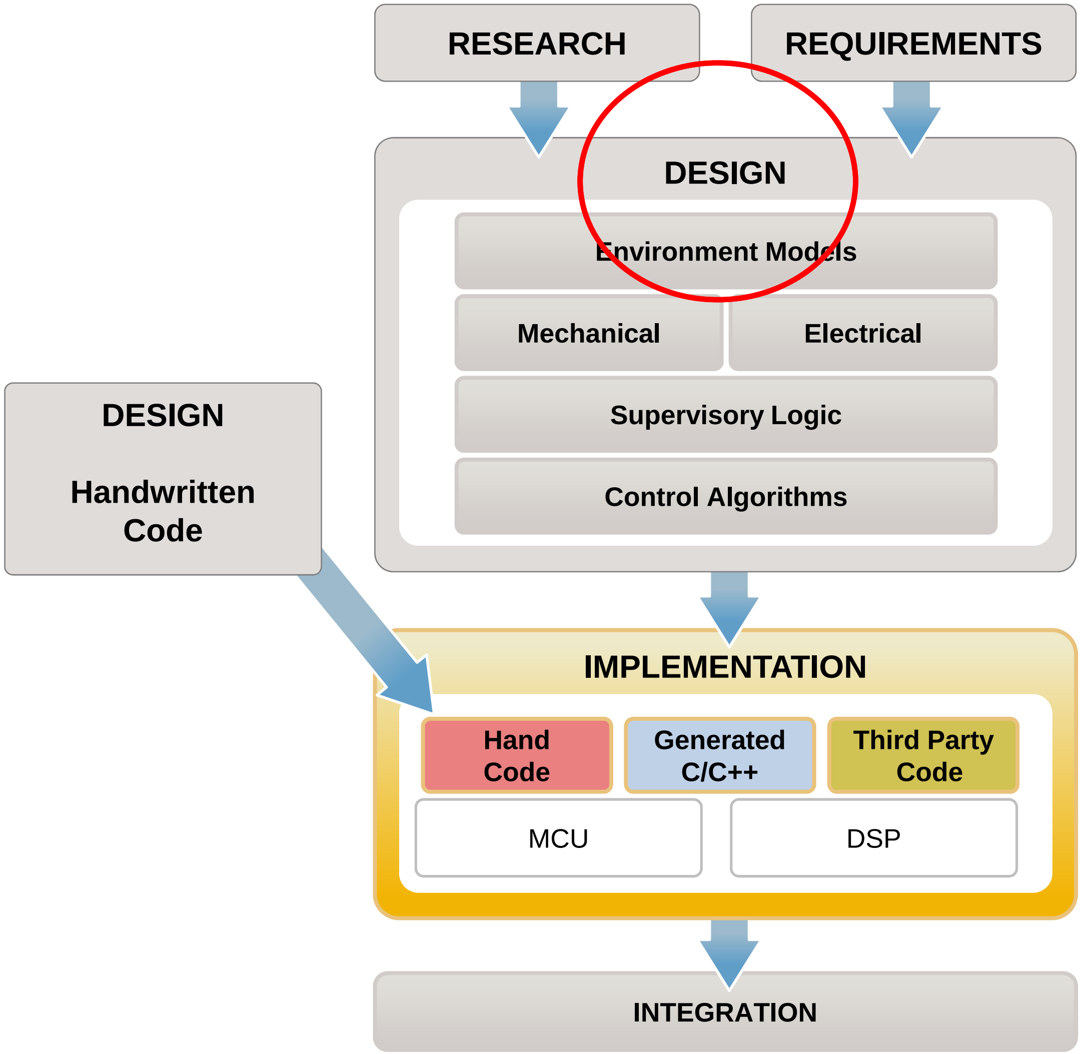


Model Verification

Code Verification

- Detect and fix design errors
  - *Robust Design*
    - Detect and fix code errors
      - *Robust Code*

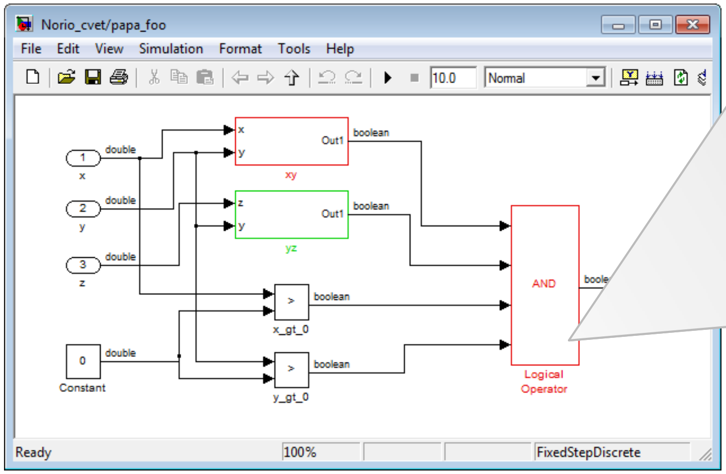
Design Error Detection in MBD

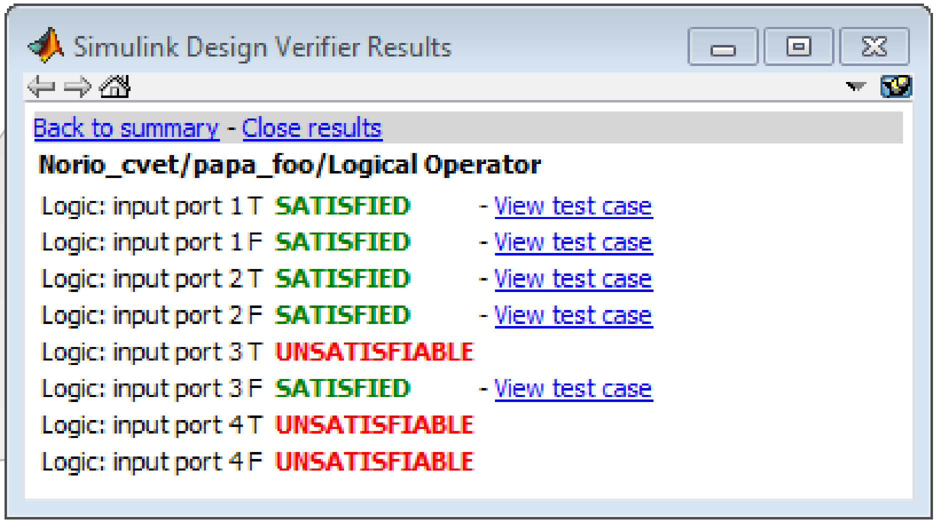


Process of Design Error Detection in MBD

- Verify design at the model level (*model verification*)
  - Identify issues such as dead logic

- Exhaustively verify design
  - Using formal methods





Back to summary - Close results

Norio_cvet/papa_foo/Logical Operator

Logic: input port 1 T **SATISFIED** - View test case
Logic: input port 1 F **SATISFIED** - View test case
Logic: input port 2 T **SATISFIED** - View test case
Logic: input port 2 F **SATISFIED** - View test case
Logic: input port 3 T **UNSATISFIABLE**
Logic: input port 3 F **SATISFIED** - View test case
Logic: input port 4 T **UNSATISFIABLE**
Logic: input port 4 F **UNSATISFIABLE**

Formal Methods

- Mathematical based techniques for
  - Specification, development and verification of software

- Proof based verification
  - Formally prove attributes of a system
  - Results are considered “*sound*”

- Example techniques
  - Model checking for exhaustive search for all states
  - Abstract interpretation for semantic analysis of programs

Introduction to Abstract Interpretation

- Formal methods based verification
  - Solution that can be applied to software programs

- What is Abstract Interpretation?
  - Consider the multiplication of three large integers

$$\boxed{-4586 \times 34985 \times 2389 = ?}$$

Application of Abstract Interpretation

- Abstract result of computation to sign domain
  - Could be positive or negative
  - Sign of the computation will be negative
- Determining sign
  - An application of Abstract Interpretation
- Technique enables precise knowledge of some properties
  - The sign, without having to multiply integers fully
  - Sign will never be positive for this computation
- Abstract Interpretation is ***sound*** and ***exhaustively proves***
  - That sign of the operation will always be negative
  - And never positive

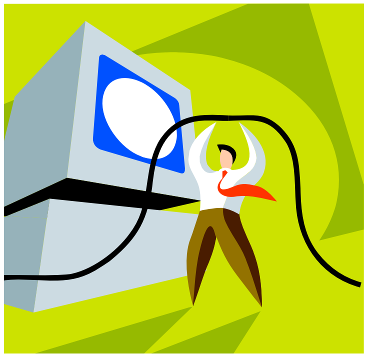
Verification Tools that Implement Model Checking and Abstract Interpretation

Verification Tools	**Reference**
ImProve for building high assurance embedded applications	Tom Hawkins
UPPAAL for modeling, validation and verification of real-time systems	Aalborg University
Stacktool for stack overflow checking of embedded software	University of Utah
DAEDALUS for validating critical software	European IST Programme
And many others …	*Search engines, Wikipedia, ….*

In this tutorial …

- We use MathWorks verification tools to demonstrate examples of applying formal methods
- To demonstrate how one can attempt to achieve zero defect software
- Applicable to any tool or product that implements formal methods

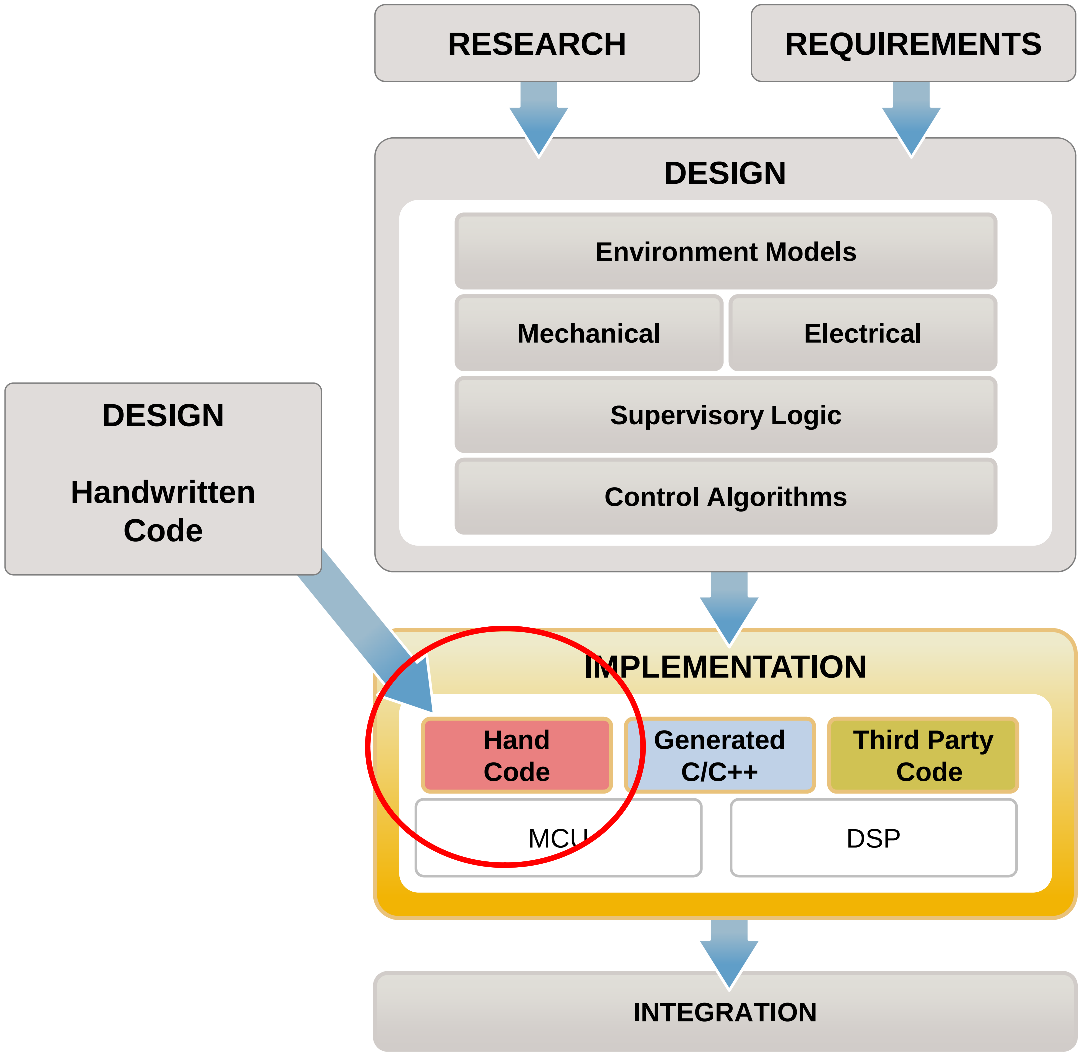
Confirming sound design



Tutorial Demo

- Design verification of a model

Verification of Handwritten Code



Typical Methods of Software Verification and Testing

- Code reviews
  - Fagan inspections to reduce coding errors
  - Process needs to be complemented with other methods

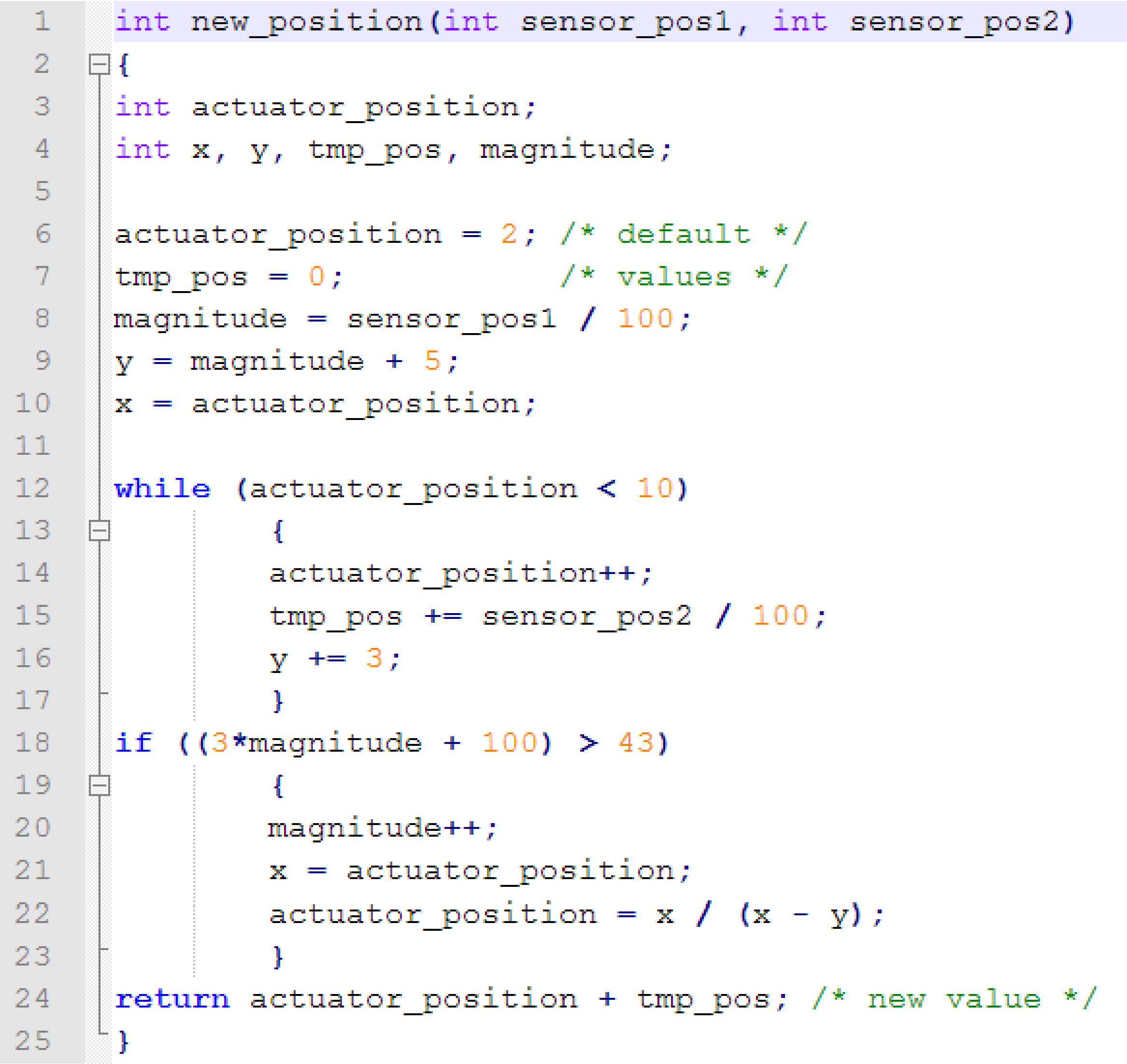
- Dynamic test
  - Validate that software meets requirements
  - Verify the execution flow of software, often on the target

When Are You Done?

- Dijkstra
  - “Program testing can be used to show the presence of bugs, but never to show their absence”

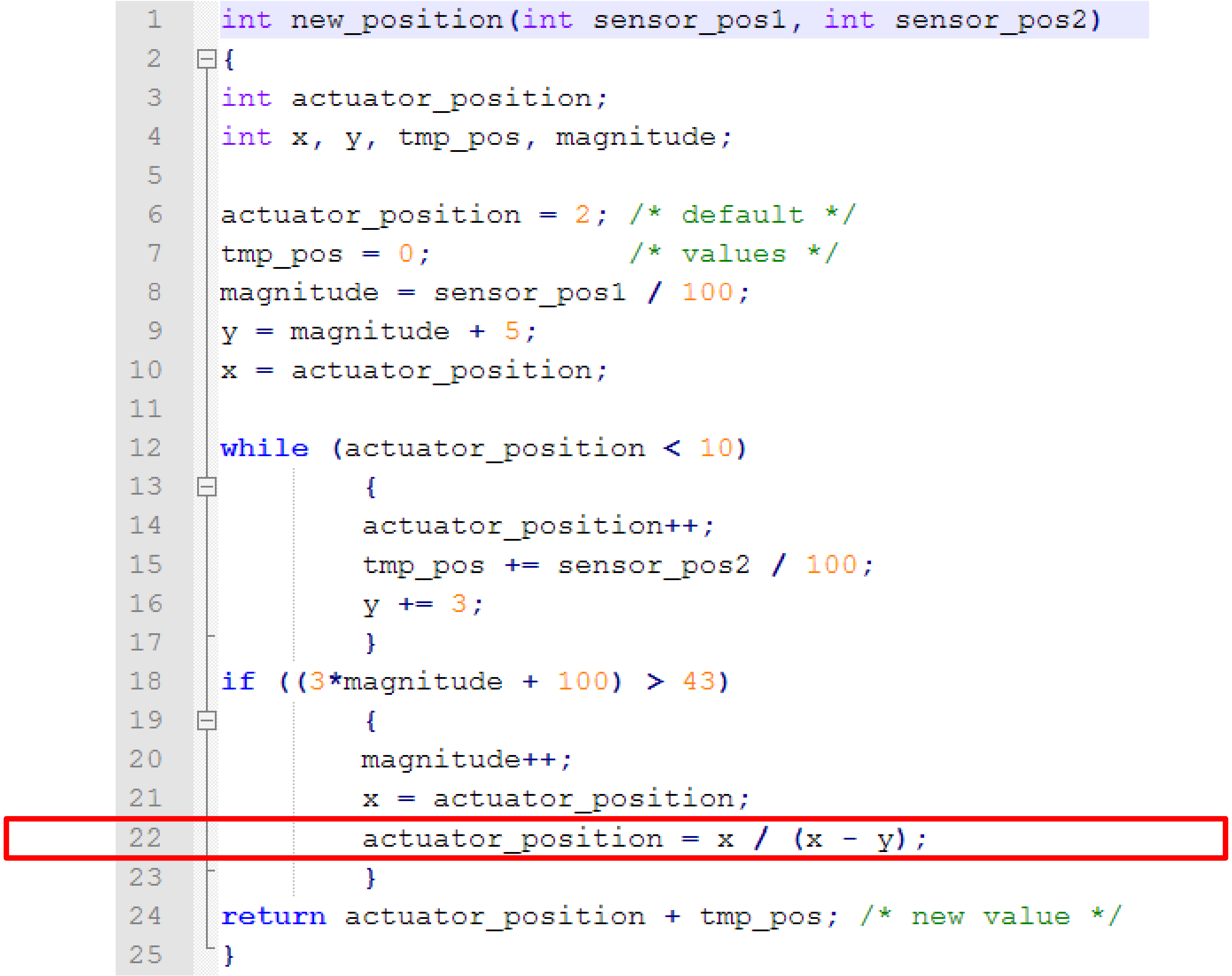
- Hailpern
  - “Given that we cannot really show there are no more errors in the program, when do we stop testing?”

Find the Run-Time Error in *new_position()*



```c
int new_position(int sensor_pos1, int sensor_pos2)
{
int actuator_position;
int x, y, tmp_pos, magnitude;

actuator_position = 2; /* default */
tmp_pos = 0;           /* values */
magnitude = sensor_pos1 / 100;
y = magnitude + 5;
x = actuator_position;

while (actuator_position < 10)
        {
        actuator_position++;
        tmp_pos += sensor_pos2 / 100;
        y += 3;
        }
if ((3*magnitude + 100) > 43)
        {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
        }
return actuator_position + tmp_pos; /* new value */
}
```

Find the Run-Time Error in *new_position()*



```
int new_position(int sensor_pos1, int sensor_pos2)
{
int actuator_position;
int x, y, tmp_pos, magnitude;

actuator_position = 2; /* default */
tmp_pos = 0;           /* values */
magnitude = sensor_pos1 / 100;
y = magnitude + 5;
x = actuator_position;

while (actuator_position < 10)
        {
        actuator_position++;
        tmp_pos += sensor_pos2 / 100;
        y += 3;
        }
if ((3*magnitude + 100) > 43)
        {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
        }
return actuator_position + tmp_pos; /* new value */
}
```

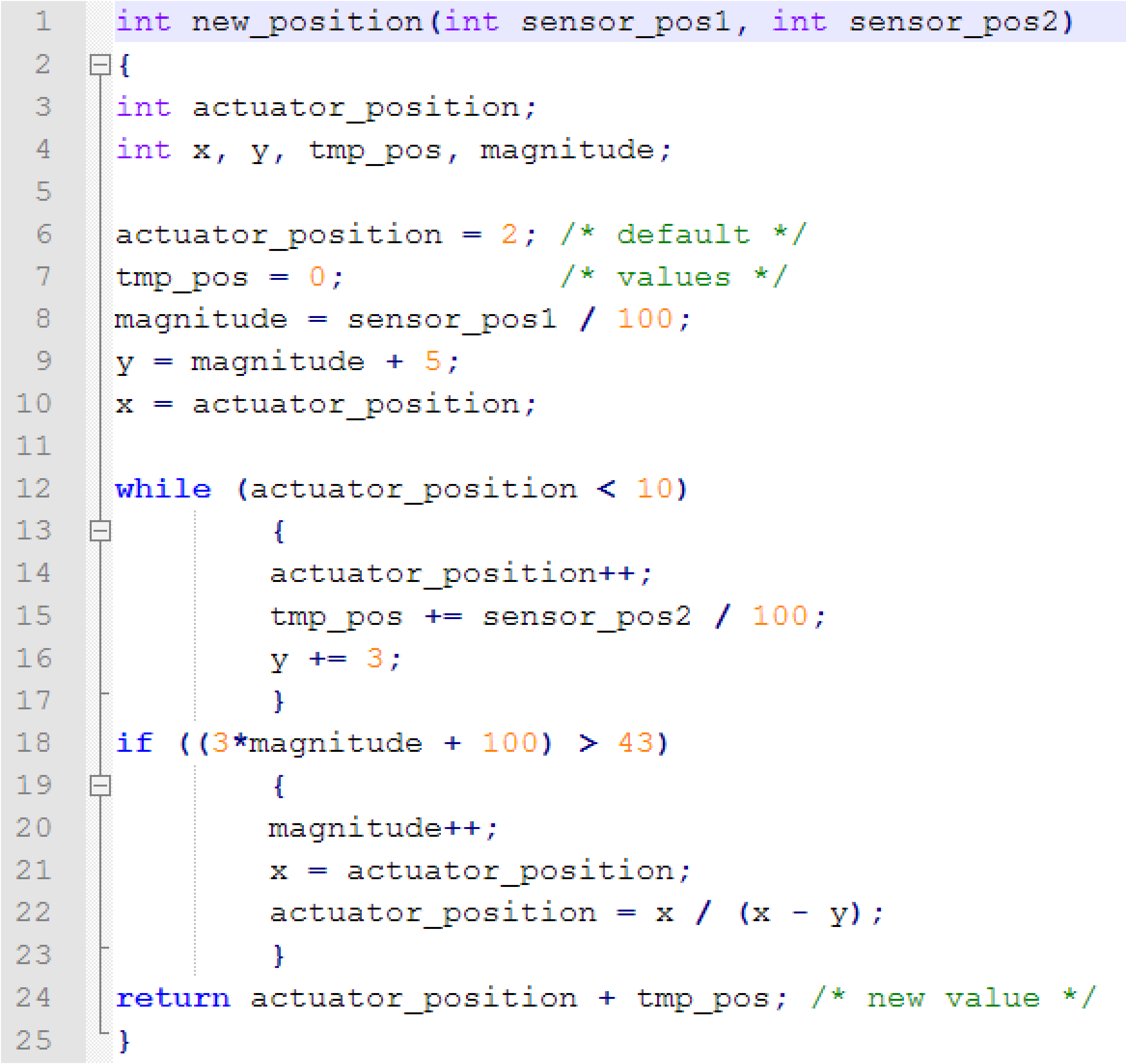
Consider the operation: `x / (x - y)`

Potential run-time errors

- Variables x and y may not be initialized
- An overflow on subtraction
- If x == y, then a divide by zero will occur

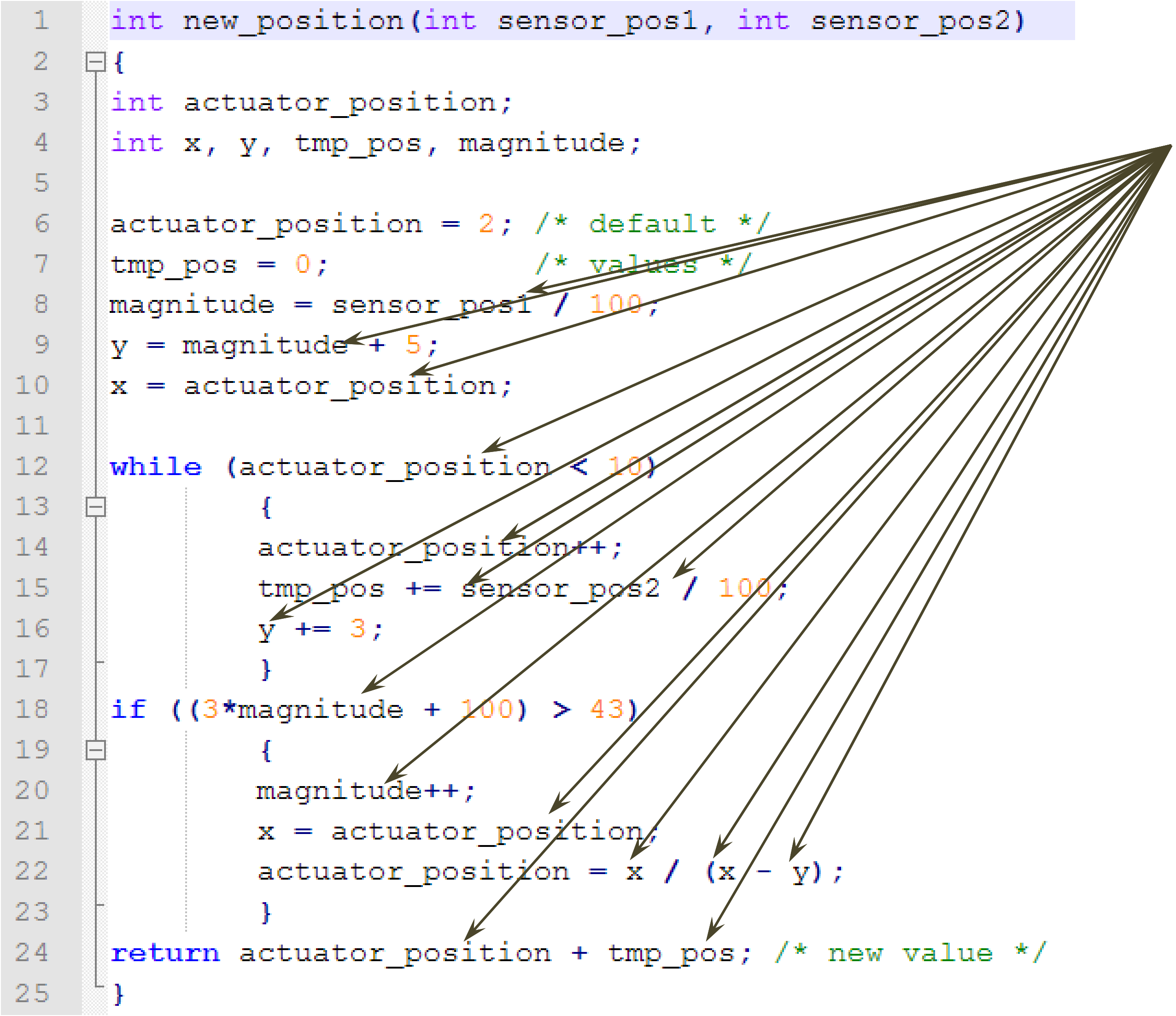
How to prove that run-time errors *do* or *do not* exist?

Code Review of *new_position()*



```
int new_position(int sensor_pos1, int sensor_pos2)
{
int actuator_position;
int x, y, tmp_pos, magnitude;

actuator_position = 2; /* default */
tmp_pos = 0;           /* values */
magnitude = sensor_pos1 / 100;
y = magnitude + 5;
x = actuator_position;

while (actuator_position < 10)
        {
        actuator_position++;
        tmp_pos += sensor_pos2 / 100;
        y += 3;
        }
if ((3*magnitude + 100) > 43)
        {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
        }
return actuator_position + tmp_pos; /* new value */
}
```

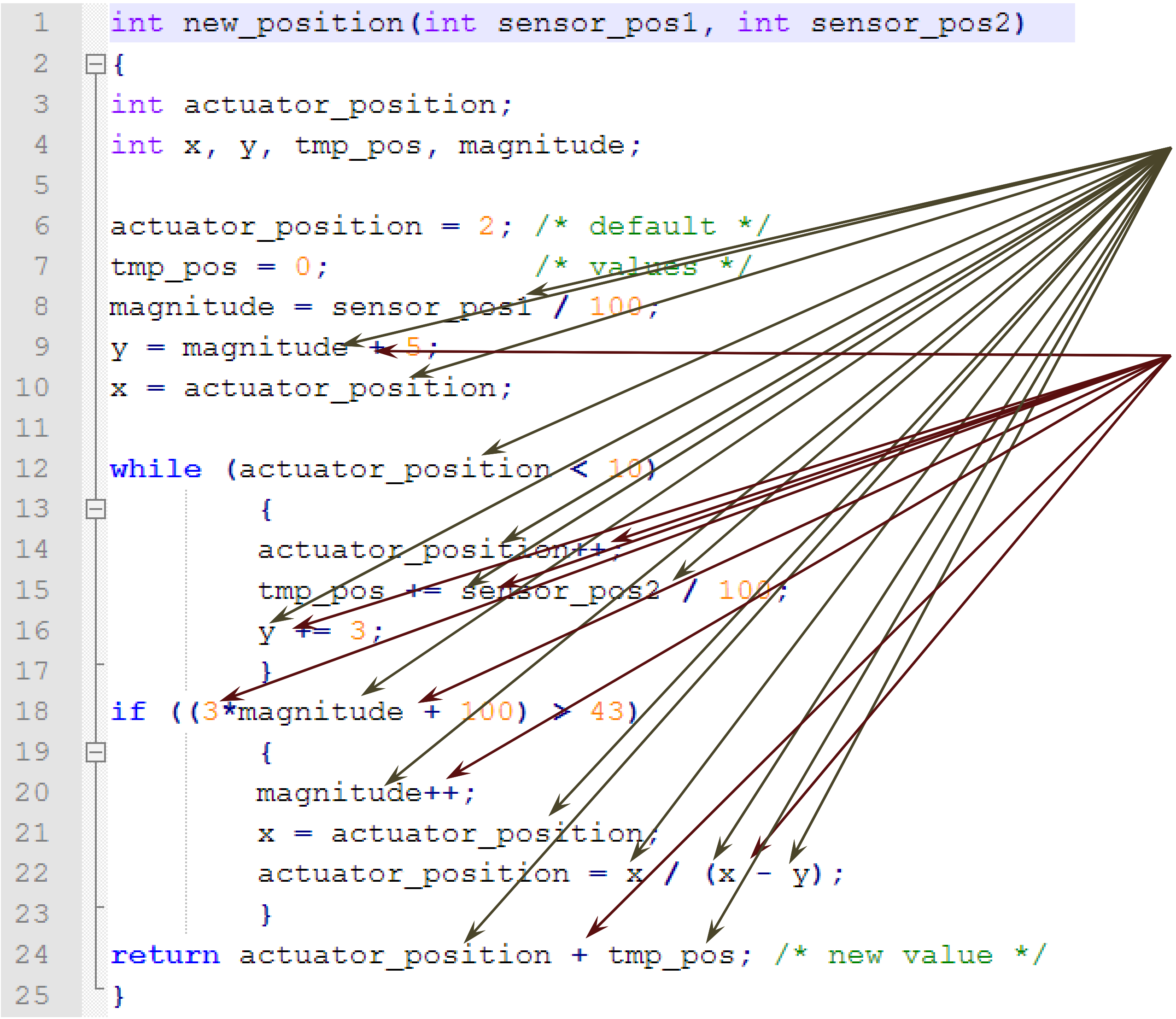
Code Review of *new_position()*



```
int new_position(int sensor_pos1, int sensor_pos2)
{
int actuator_position;
int x, y, tmp_pos, magnitude;

actuator_position = 2; /* default */
tmp_pos = 0;           /* values */
magnitude = sensor_pos1 / 100;
y = magnitude + 5;
x = actuator_position;

while (actuator_position < 10)
        {
        actuator_position++;
        tmp_pos += sensor_pos2 / 100;
        y += 3;
        }
if ((3*magnitude + 100) > 43)
        {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
        }
return actuator_position + tmp_pos; /* new value */
}
```

Variables may not be initialized

Code Review of *new_position()*

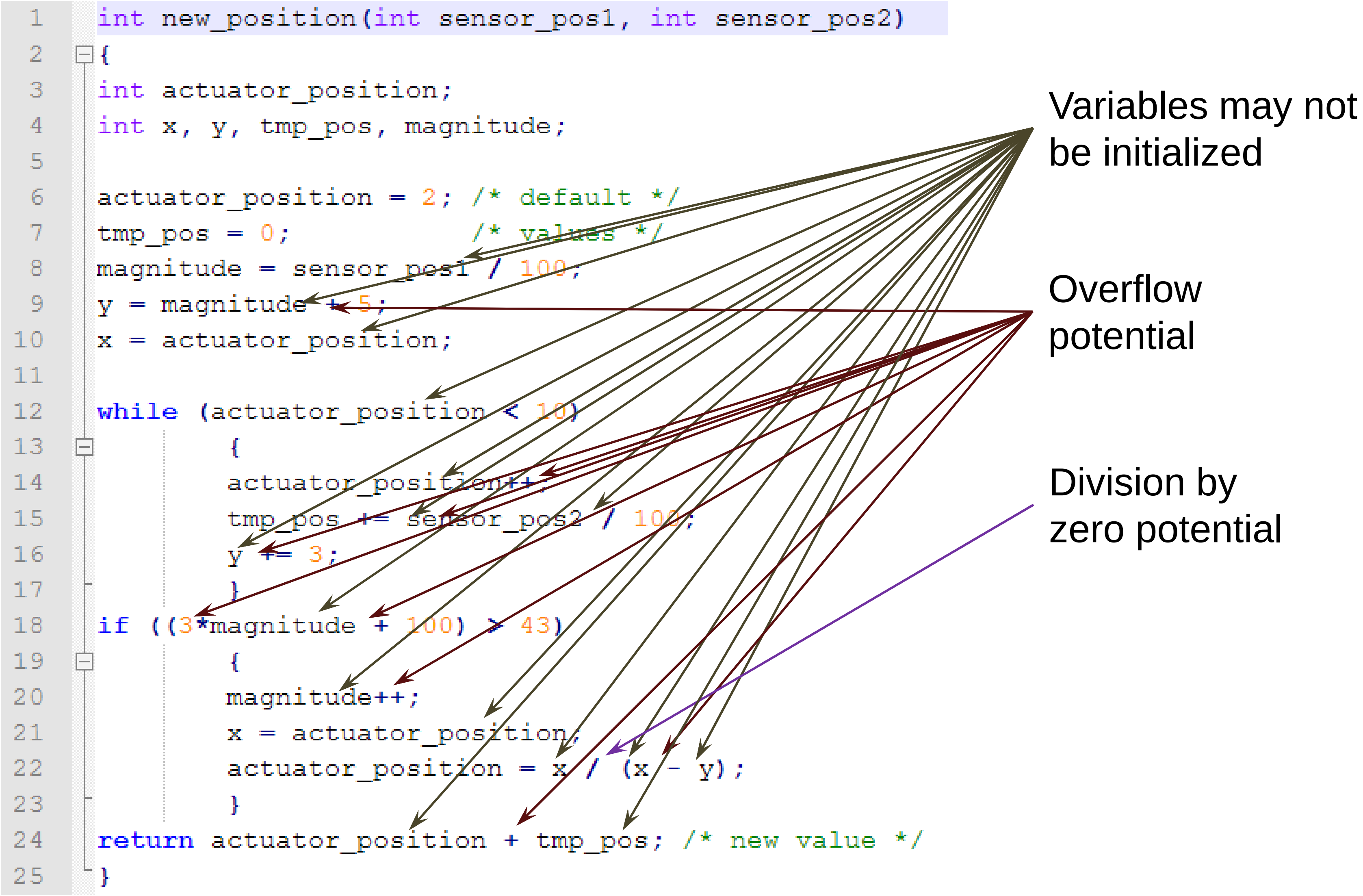


```
int new_position(int sensor_pos1, int sensor_pos2)
{
int actuator_position;
int x, y, tmp_pos, magnitude;

actuator_position = 2; /* default */
tmp_pos = 0;           /* values */
magnitude = sensor_pos1 / 100;
y = magnitude + 5;
x = actuator_position;

while (actuator_position < 10)
        {
        actuator_position++;
        tmp_pos += sensor_pos2 / 100;
        y += 3;
        }
if ((3*magnitude + 100) > 43)
        {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
        }
return actuator_position + tmp_pos; /* new value */
}
```

Variables may not be initialized

Overflow potential

Code Review of *new_position()*



```
int new_position(int sensor_pos1, int sensor_pos2)
{
int actuator_position;
int x, y, tmp_pos, magnitude;

actuator_position = 2; /* default */
tmp_pos = 0;           /* values */
magnitude = sensor_pos1 / 100;
y = magnitude + 5;
x = actuator_position;

while (actuator_position < 10)
        {
        actuator_position++;
        tmp_pos += sensor_pos2 / 100;
        y += 3;
        }
if ((3*magnitude + 100) > 43)
        {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
        }
return actuator_position + tmp_pos; /* new value */
}
```

Variables may not be initialized

Overflow potential

Division by zero potential

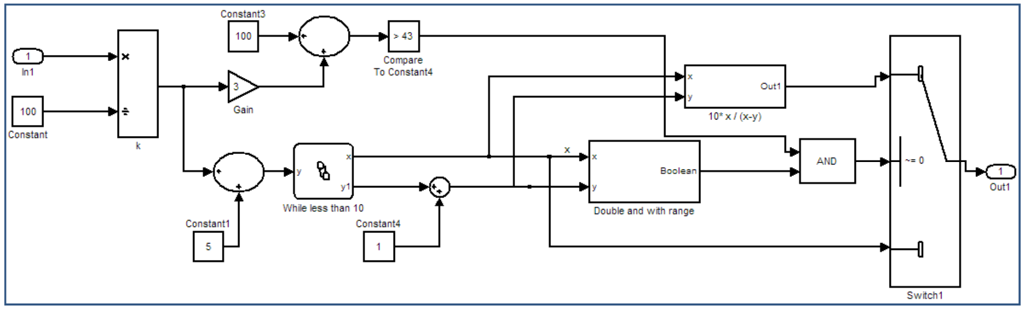
Code Review and Dynamic Test

- Code review results
  - Initially identified potential divide by zero condition
  - Deeper review shows potential overflow and initialization issues
- Next step is to Test
  - Validate that code written to meet requirements
  - Verify that the code is robust and will not fail

Requirements Specification

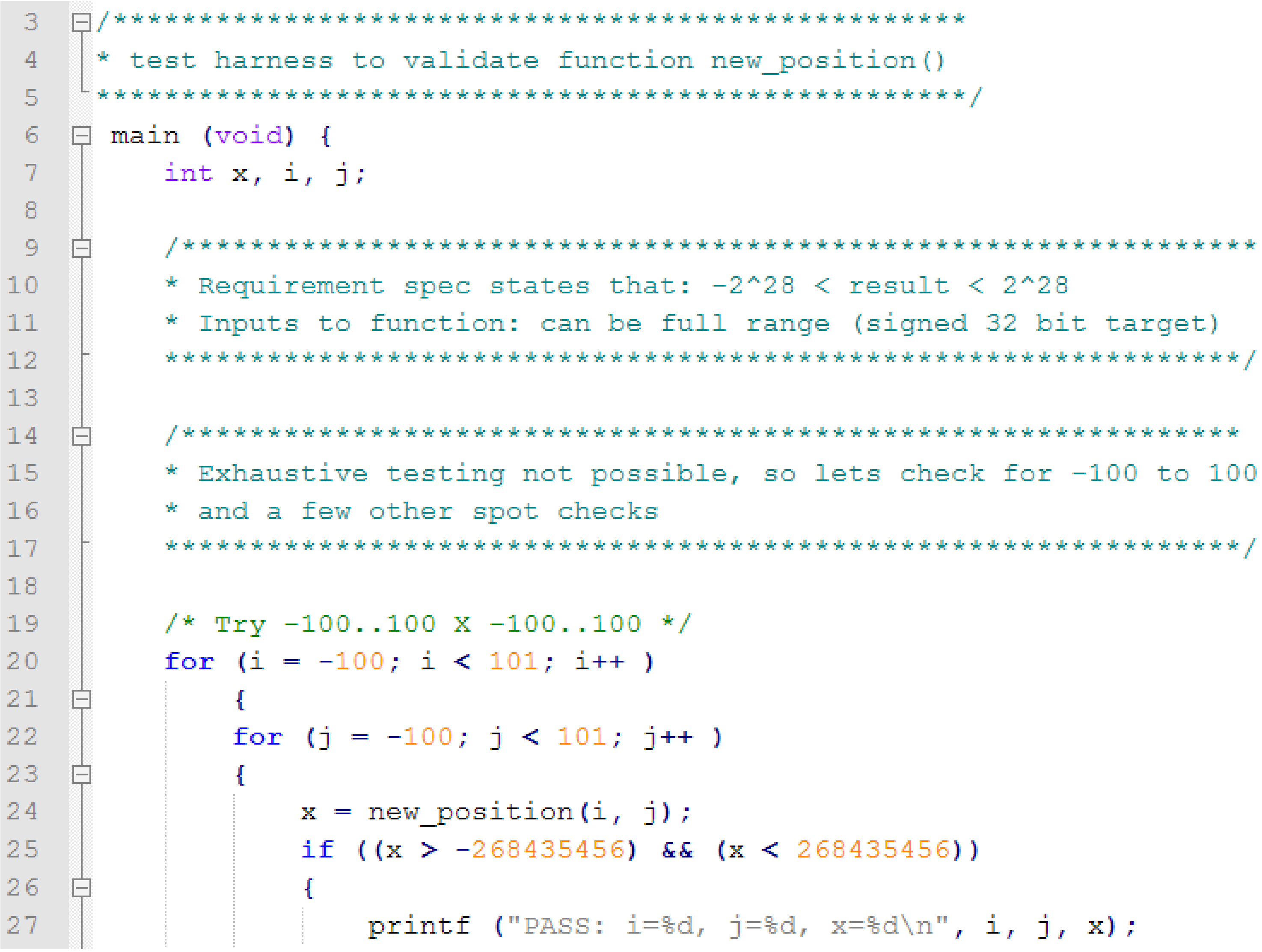
- Compute new position of control arm based on 2 position sensors

- Implement algorithm as modeled in the Simulink modeling environment



- Return value of new position shall be within ± 2^{28}

Dynamic Test with a Test-Harness



```
/***************************************************
* test harness to validate function new_position()
***************************************************/
main (void) {
    int x, i, j;

    /***************************************************************
    * Requirement spec states that: -2^28 < result < 2^28
    * Inputs to function: can be full range (signed 32 bit target)
    ***************************************************************/

    /***************************************************************
    * Exhaustive testing not possible, so lets check for -100 to 100
    * and a few other spot checks
    ***************************************************************/

    /* Try -100..100 X -100..100 */
    for (i = -100; i < 101; i++ )
        {
        for (j = -100; j < 101; j++ )
        {
            x = new_position(i, j);
            if ((x > -268435456) && (x < 268435456))
            {
                printf ("PASS: i=%d, j=%d, x=%d\n", i, j, x);
```

Exhaustive Testing of *new_position()*

- Both inputs are signed int32
  - Full range inputs: `-2`^{31}`-1 .. +2`^{31}`-1`
  - All combinations of two inputs: 4.61X10^{18} test-cases
- Test time on a Windows host machine
  - 2.2GHz T7500 Intel processor
  - 4 million test-cases took 9.284 seconds
  - Exhaustive testing time: **339,413 years**

Exhaustive Testing is Impossible

How to Increase Confidence?

- Could do more spot testing
  - But it is still not exhaustive
- Add defensive code (*if x != y …*)
  - This will protect against divide by zero!
  - But adds more code and execution overhead
  - What about other potential errors like overflow?
- Wish that the code will not fail
  - Is that a good strategy …
- What about static code analysis tools?
  - Compiler warnings and more sophisticated tools

Introduction to Static Code Analysis

- Scanning source code to automate software verification
- Range from ***unsound*** methods to ***sound*** techniques

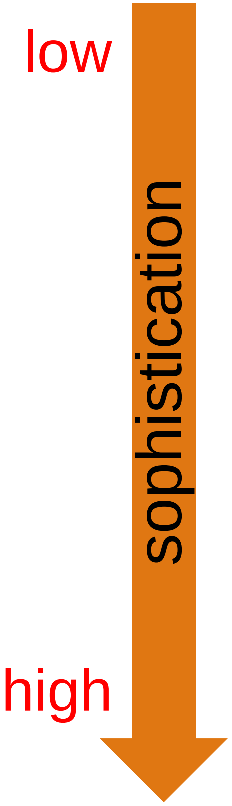
Introduction to Static Code Analysis

- Scanning source code to automate software verification
- Range from ***unsound*** methods to ***sound*** techniques

low
sophistication
high

Introduction to Static Code Analysis

- Scanning source code to automate software verification
- Range from ***unsound*** methods to ***sound*** techniques



Compiler warnings

- Incompatible type detection, etc.

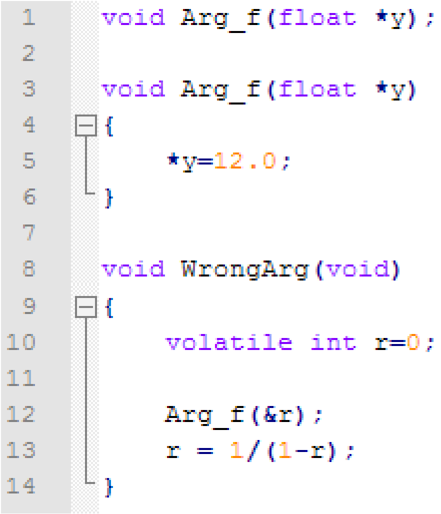
Bug finding

- Pattern matching, heuristics, data/control flow

Formal methods

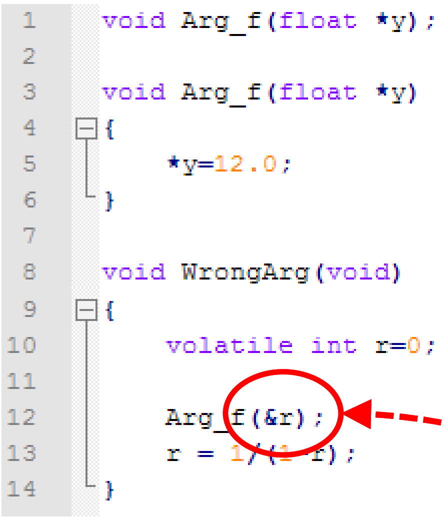
- Sound proof based techniques, applied to source code

Compiler Warning Example

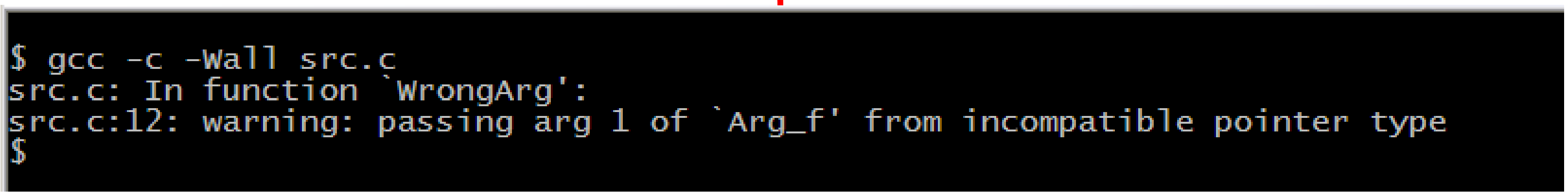


```c
void Arg_f(float *y);

void Arg_f(float *y)
{
    *y=12.0;
}

void WrongArg(void)
{
    volatile int r=0;

    Arg_f(&r);
    r = 1/(1-r);
}
```

Compiler Warning Example

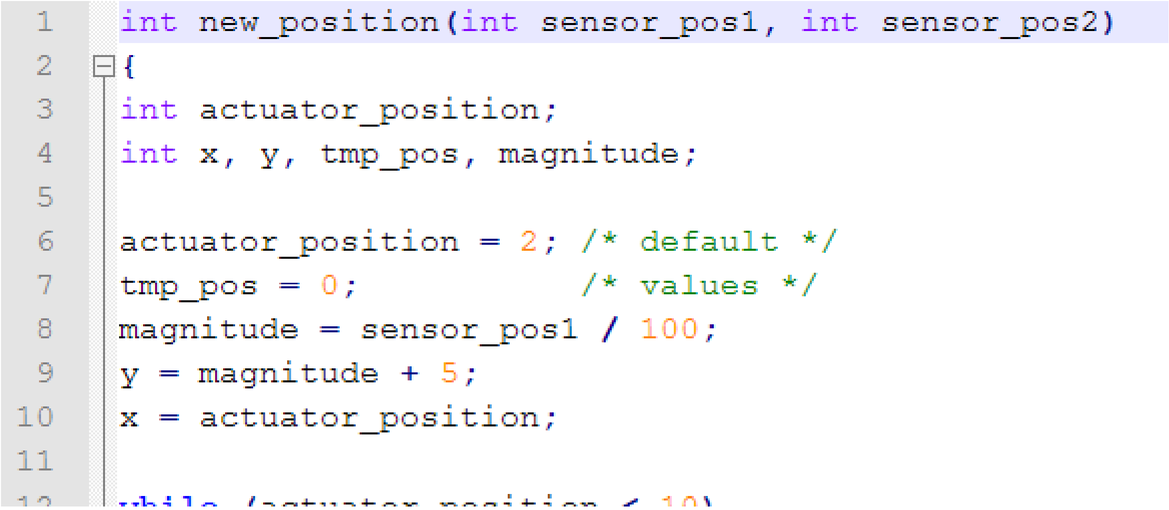


```
 1  void Arg_f(float *y);
 2
 3  void Arg_f(float *y)
 4  {
 5      *y=12.0;
 6  }
 7
 8  void WrongArg(void)
 9  {
10      volatile int r=0;
11
12      Arg_f(&r);
13      r = 1/(1-r);
14  }
```



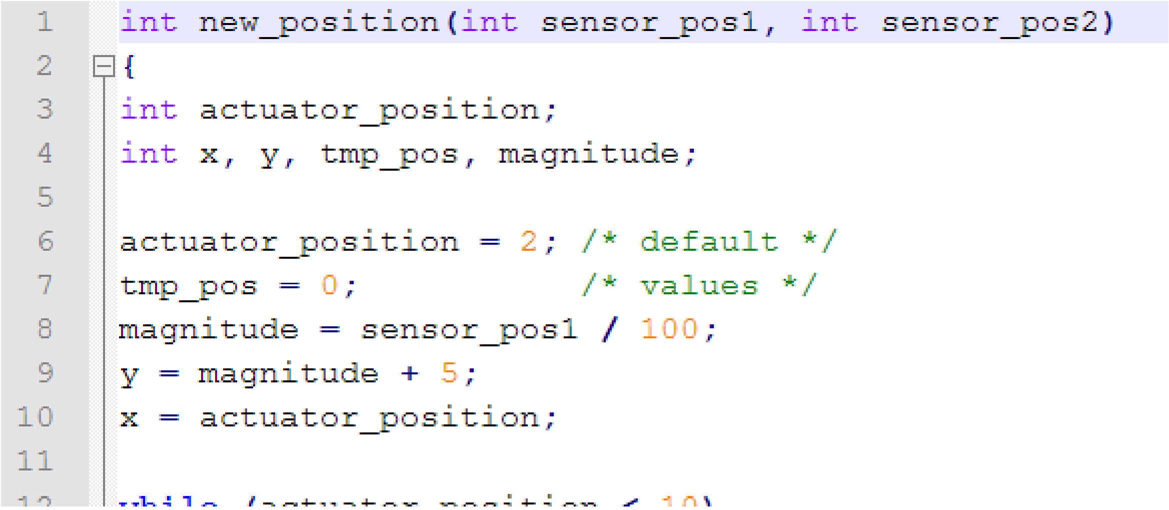
```
$ gcc -c -Wall src.c
src.c: In function `WrongArg':
src.c:12: warning: passing arg 1 of `Arg_f' from incompatible pointer type
$
```

Compiler Warnings for *new_position()*

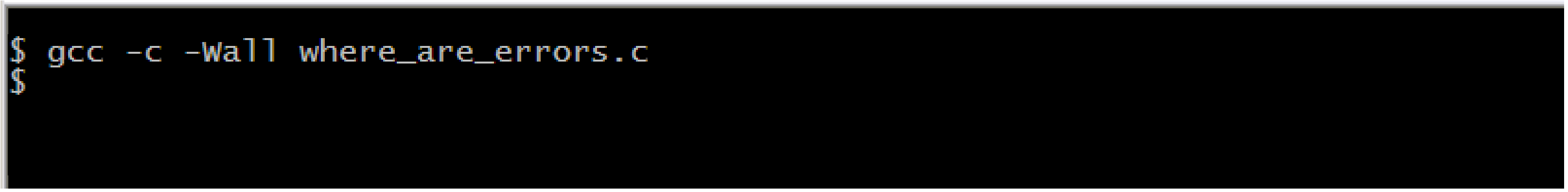


```
int new_position(int sensor_pos1, int sensor_pos2)
{
int actuator_position;
int x, y, tmp_pos, magnitude;

actuator_position = 2; /* default */
tmp_pos = 0;           /* values */
magnitude = sensor_pos1 / 100;
y = magnitude + 5;
x = actuator_position;

while (actuator_position < 10)
```

Compiler Warnings for *new_position()*

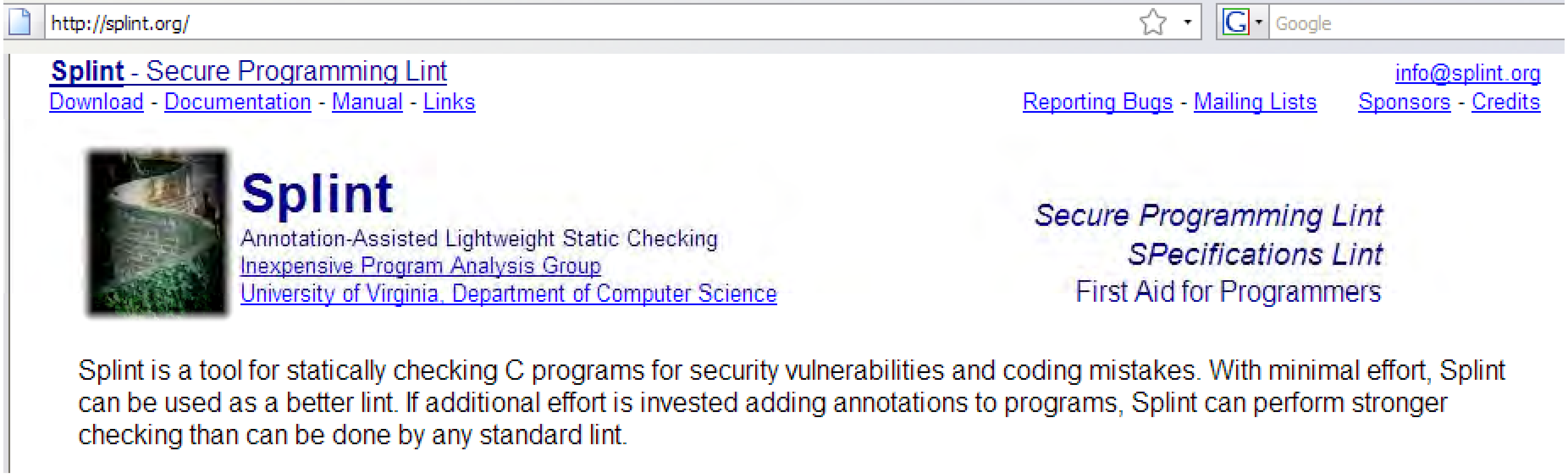


```
int new_position(int sensor_pos1, int sensor_pos2)
{
int actuator_position;
int x, y, tmp_pos, magnitude;

actuator_position = 2; /* default */
tmp_pos = 0;           /* values */
magnitude = sensor_pos1 / 100;
y = magnitude + 5;
x = actuator_position;

while (actuator_position < 10)
```



```
$ gcc -c -Wall where_are_errors.c
$
```

Static Analysis with Splint (*splint.org*)



Splint - Secure Programming Lint

Download - Documentation - Manual - Links

info@splint.org

Reporting Bugs - Mailing Lists   Sponsors - Credits

Splint

Annotation-Assisted Lightweight Static Checking

Inexpensive Program Analysis Group

University of Virginia, Department of Computer Science

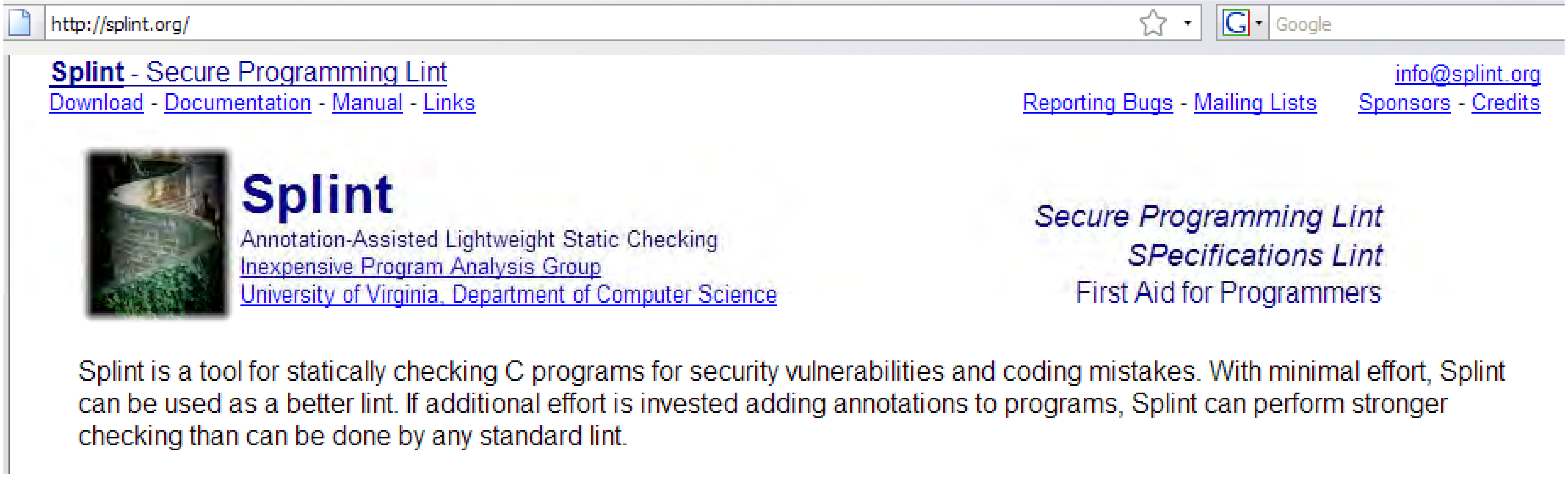
Secure Programming Lint

SPecifications Lint

First Aid for Programmers

Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes. With minimal effort, Splint can be used as a better lint. If additional effort is invested adding annotations to programs, Splint can perform stronger checking than can be done by any standard lint.

Static Analysis with Splint (*splint.org*)



Splint - Secure Programming Lint

Download - Documentation - Manual - Links

info@splint.org

Reporting Bugs - Mailing Lists   Sponsors - Credits

Splint

Annotation-Assisted Lightweight Static Checking

Inexpensive Program Analysis Group

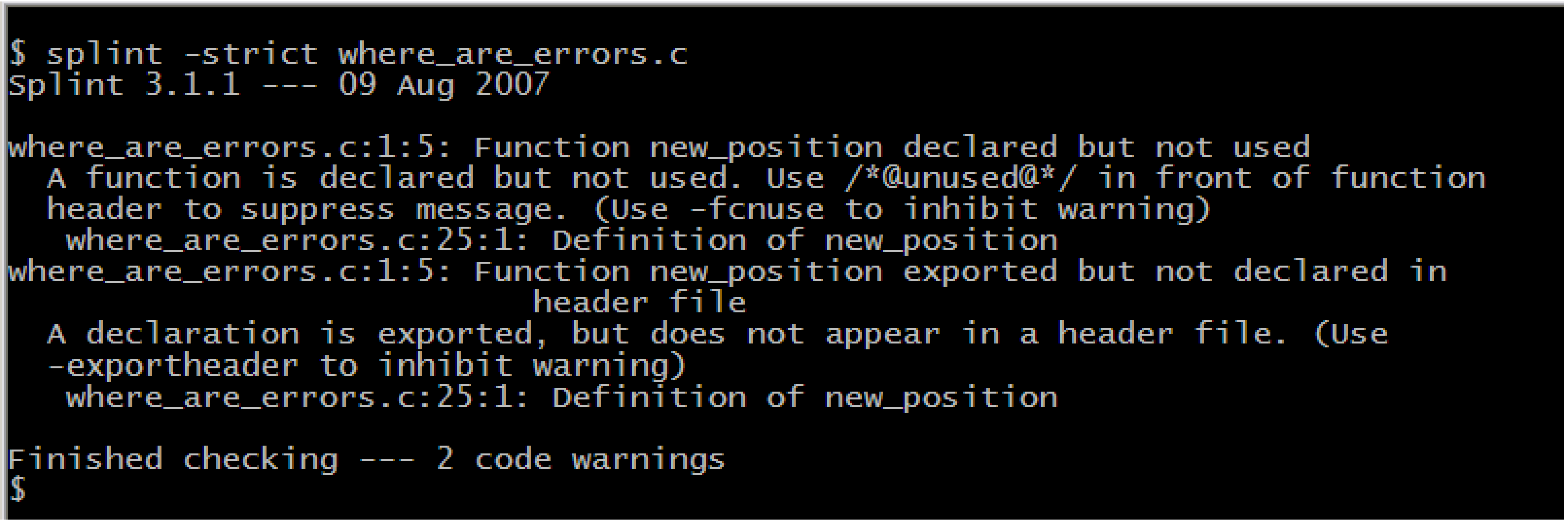
University of Virginia, Department of Computer Science

Secure Programming Lint

SPecifications Lint

First Aid for Programmers

Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes. With minimal effort, Splint can be used as a better lint. If additional effort is invested adding annotations to programs, Splint can perform stronger checking than can be done by any standard lint.



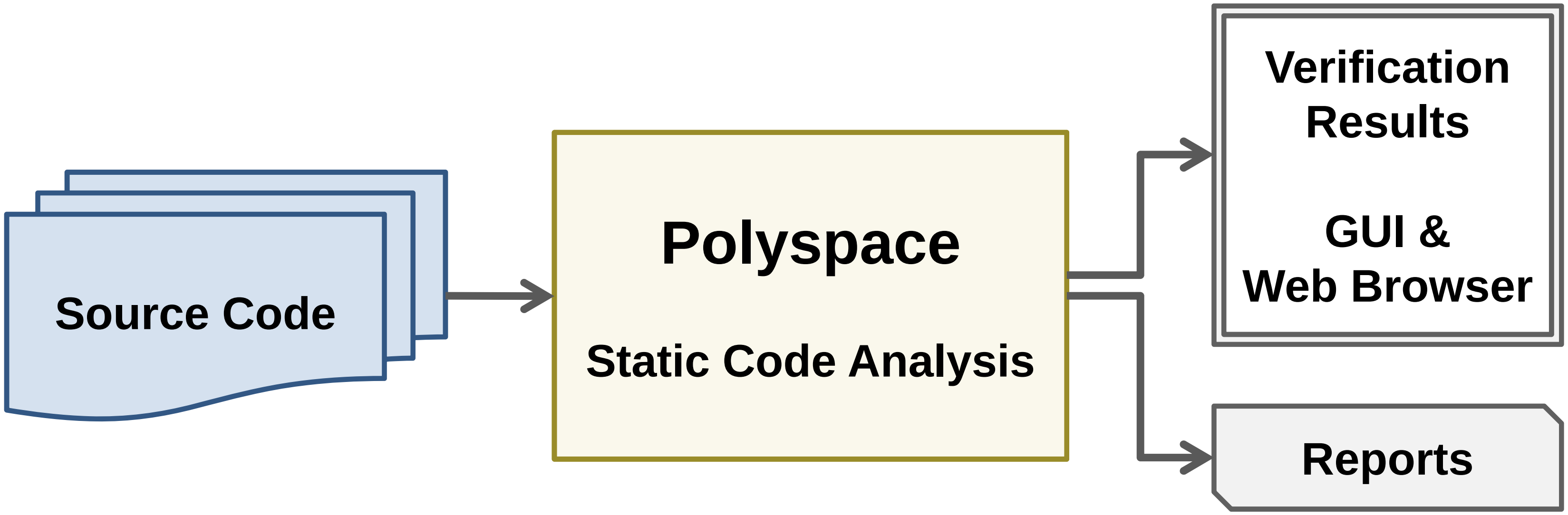
```
$ splint -strict where_are_errors.c
Splint 3.1.1 --- 09 Aug 2007

where_are_errors.c:1:5: Function new_position declared but not used
  A function is declared but not used. Use /*@unused@*/ in front of function
  header to suppress message. (Use -fcnuse to inhibit warning)
   where_are_errors.c:25:1: Definition of new_position
where_are_errors.c:1:5: Function new_position exported but not declared in
                           header file
  A declaration is exported, but does not appear in a header file. (Use
  -exportheader to inhibit warning)
   where_are_errors.c:25:1: Definition of new_position

Finished checking --- 2 code warnings
$
```

Verification Results on *new_position()*

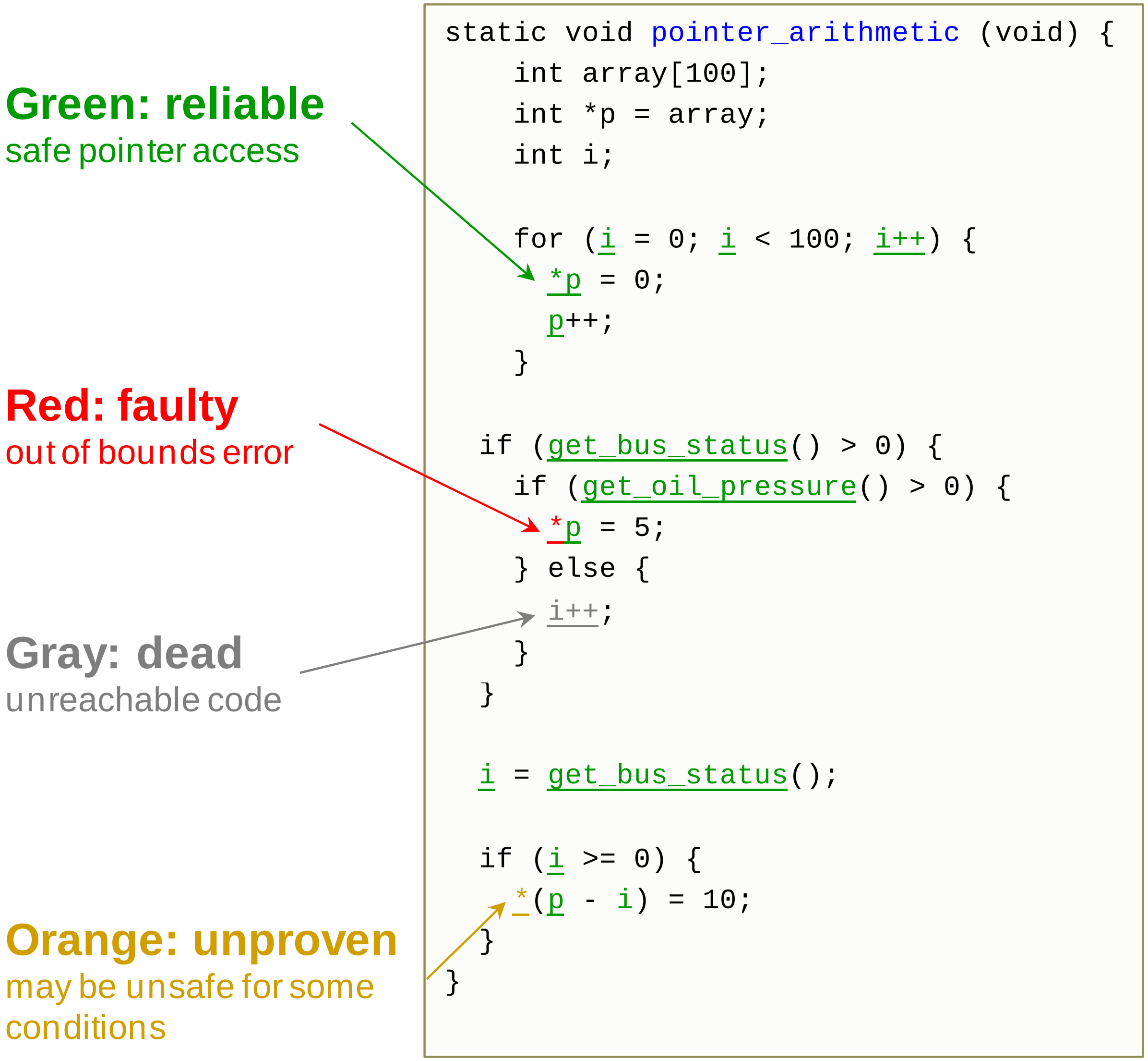
<table>
<tr><th colspan="3"><i>Required Checks</i></th></tr>
<tr><th>Activity</th><th>Comments</th><th>Status</th></tr>
<tr><td>Code Review</td><td>Identified potential non-initialized variables, overflows, and divide by zero</td><td>Further examination required</td></tr>
<tr><td>Dynamic Test</td><td>Test to requirements</td><td>Pass</td></tr>
<tr><th colspan="3"><i>Additional Confidence Checks</i></th></tr>
<tr><th>Activity</th><th>Comments</th><th>Status</th></tr>
<tr><td>Compiler warnings</td><td>None</td><td>No issues</td></tr>
<tr><td>Static Code Analysis</td><td>Splint with –strict</td><td>No issues</td></tr>
<tr><td>Formal methods</td><td></td><td></td></tr>
</table>

Formal Methods Based Static Code Analysis

- Detects and proves the absence of certain run-time errors
- Operates at source code level



Polyspace Static Code Analysis Results



Green: reliable
safe pointer access

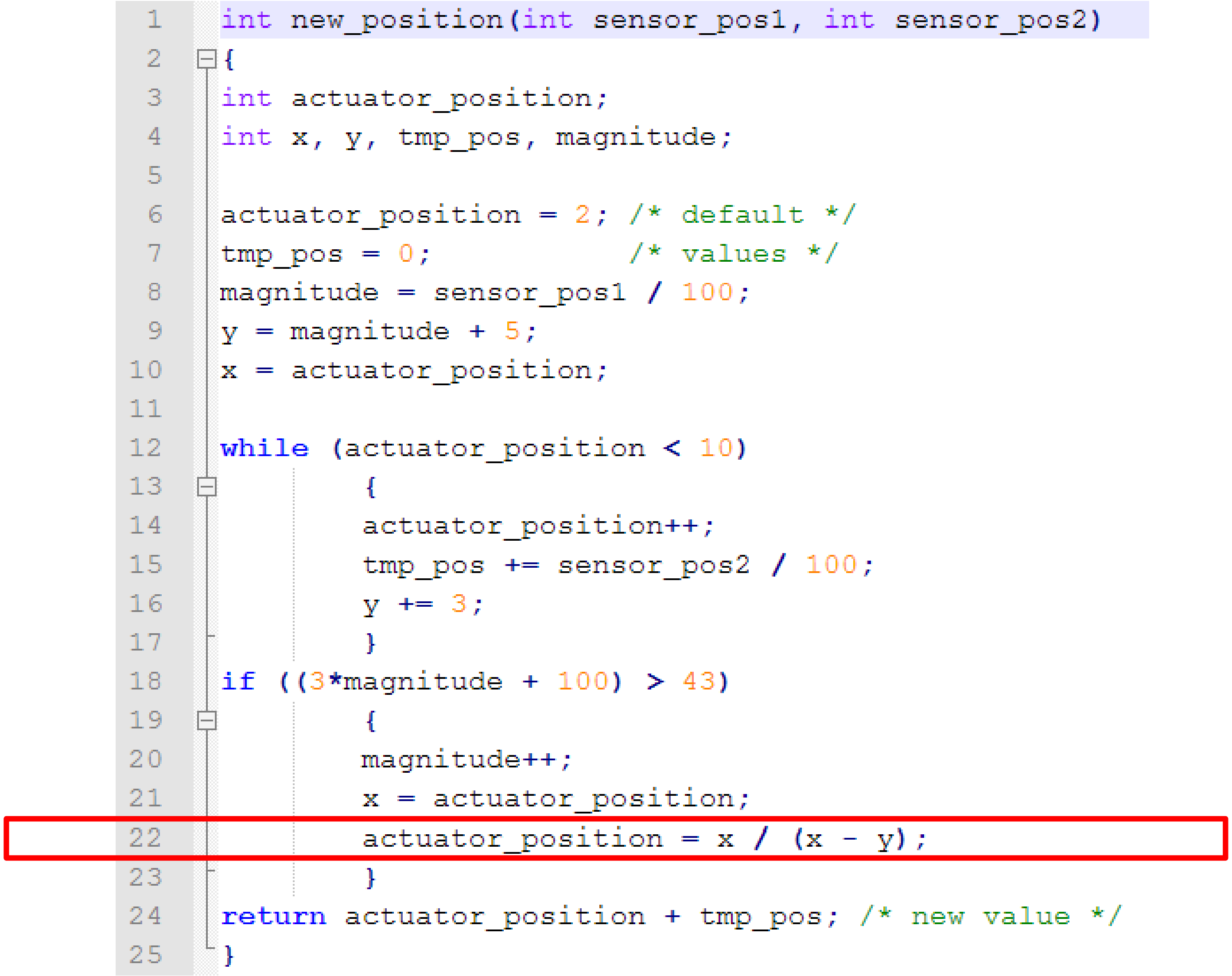
Red: faulty
out of bounds error

Gray: dead
unreachable code

Orange: unproven
may be unsafe for some conditions

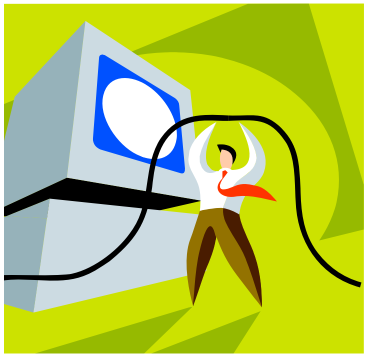
```
static void pointer_arithmetic (void) {
  int array[100];
  int *p = array;
  int i;

  for (i = 0; i < 100; i++) {
   *p = 0;
   p++;
  }

 if (get_bus_status() > 0) {
  if (get_oil_pressure() > 0) {
   *p = 5;
  } else {
   i++;
  }
 }

 i = get_bus_status();

 if (i >= 0) {
  *(p - i) = 10;
 }
}
```

Returning to our Example *new_position()*



```
int new_position(int sensor_pos1, int sensor_pos2)
{
int actuator_position;
int x, y, tmp_pos, magnitude;

actuator_position = 2; /* default */
tmp_pos = 0;           /* values */
magnitude = sensor_pos1 / 100;
y = magnitude + 5;
x = actuator_position;

while (actuator_position < 10)
        {
        actuator_position++;
        tmp_pos += sensor_pos2 / 100;
        y += 3;
        }
if ((3*magnitude + 100) > 43)
        {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
        }
return actuator_position + tmp_pos; /* new value */
}
```

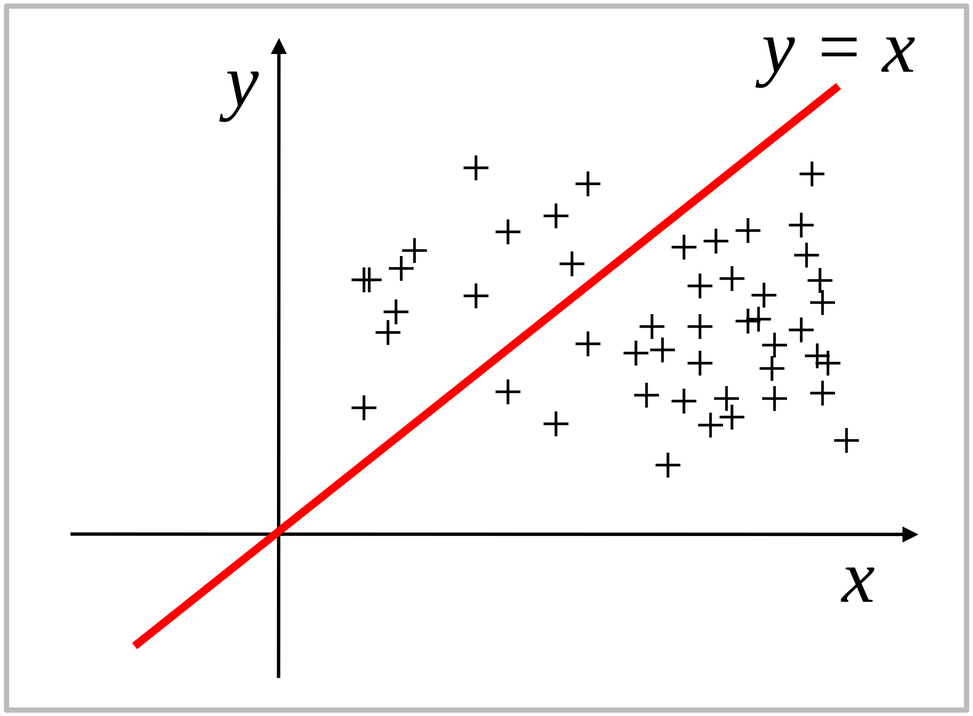
Polyspace Results on *new_position()*



Tutorial Demo

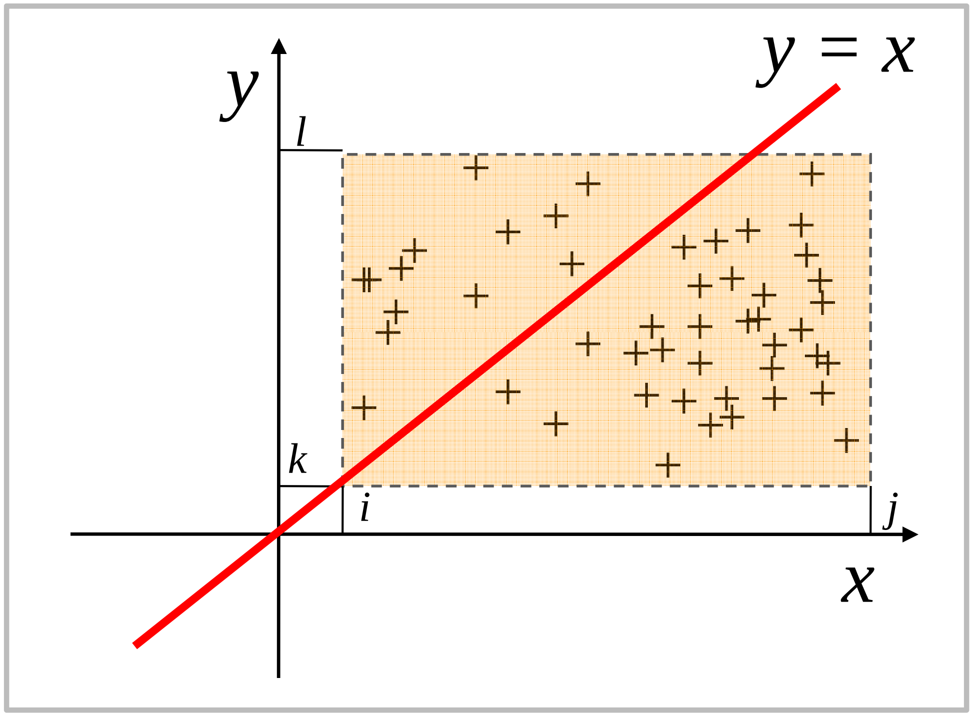
- Verification results for *new_position()*
- Results for *new_position()* with added protection

How to Prove x!=y for x/(x-y)



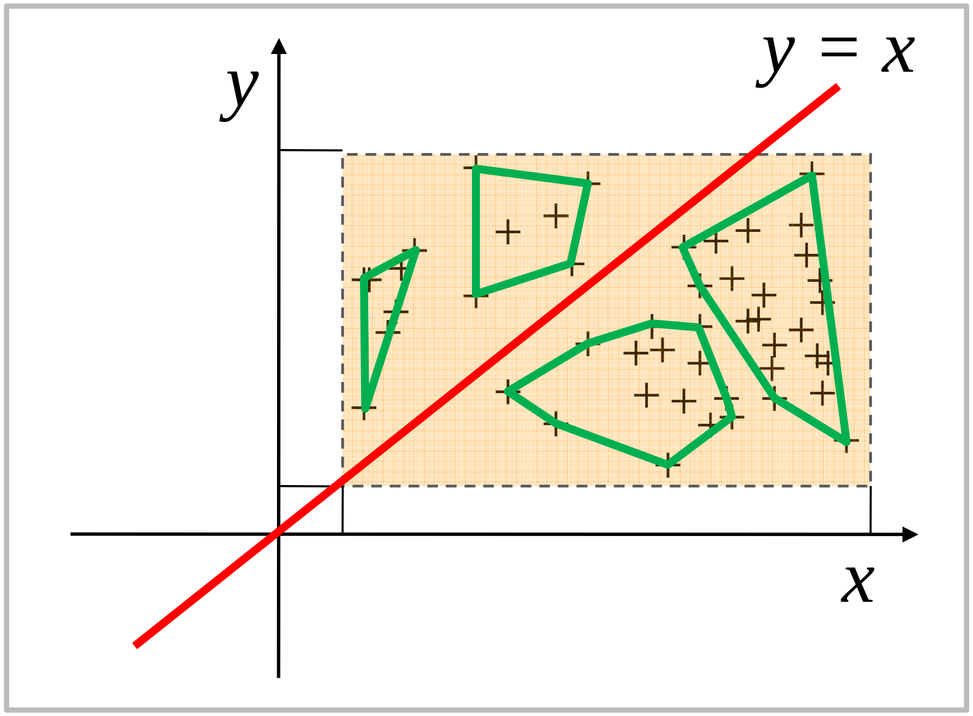
How to Prove `x!=y` for `x/(x-y)`

Type Analysis (bounding conditions)



How to Prove `x!=y` for `x/(x-y)`

With Abstract Interpretation



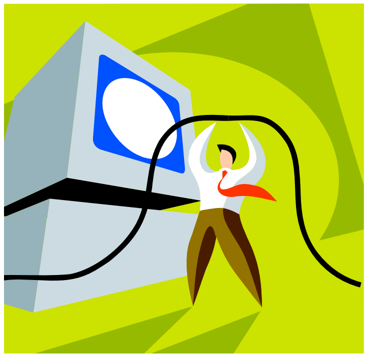
- No code execution
- No test-cases

- Exhaustive*!*
- Proven*!*

Advantages and Disadvantages

- Advantages
  - Deep formal methods based code verification
  - Can formally prove that code is defect free
    and formally prove absolute existence of a defect
  - Sound technique … identifies all potential failure points
- Disadvantages
  - Compute intensive, will take time to run
  - In practice limited to projects with <1 MLOC
  - If results are viewed conservatively,
    all potential defects must be reviewed

Verifying Complex Handwritten Code



Tutorial Demo

- Identifying run-time errors (reds)
- Dead code (grays)
- Understanding potentially failing code (oranges)
- Analysis of multithreaded coded

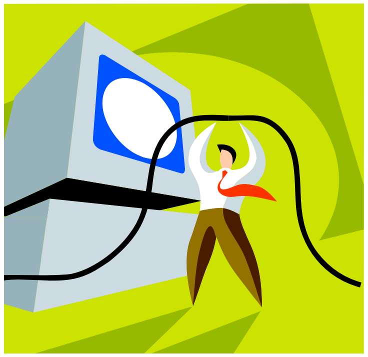
Range Violation Detection

- Some applications assume certain variable range
  - E.g. angle in degrees must be between 0 and 359
  - May simplify simulation and test

- What happens if range is violated?

- How to detect range violations exhaustively?

Range Violation Detection

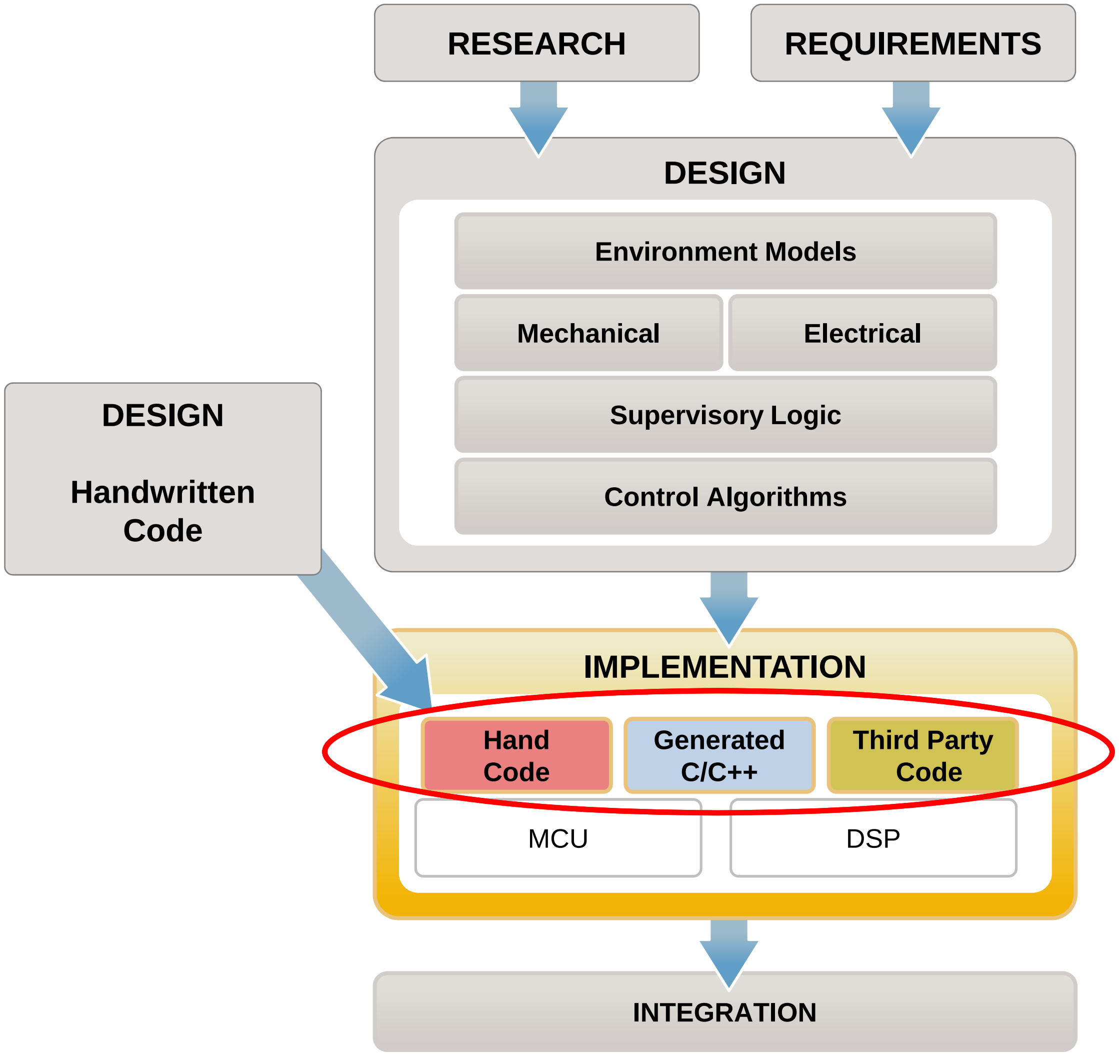


- Range violation detection

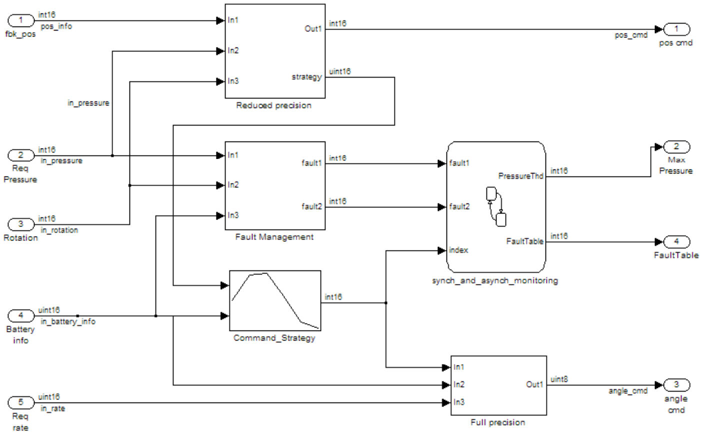
Practical Considerations of Implementing and Verifying Complex Systems

Context of automatic code generation from Model Based Design (MBD) and the reality of mixed model and code environments

Verification of a System



Returning to our Engine Controller



Complex Algorithm

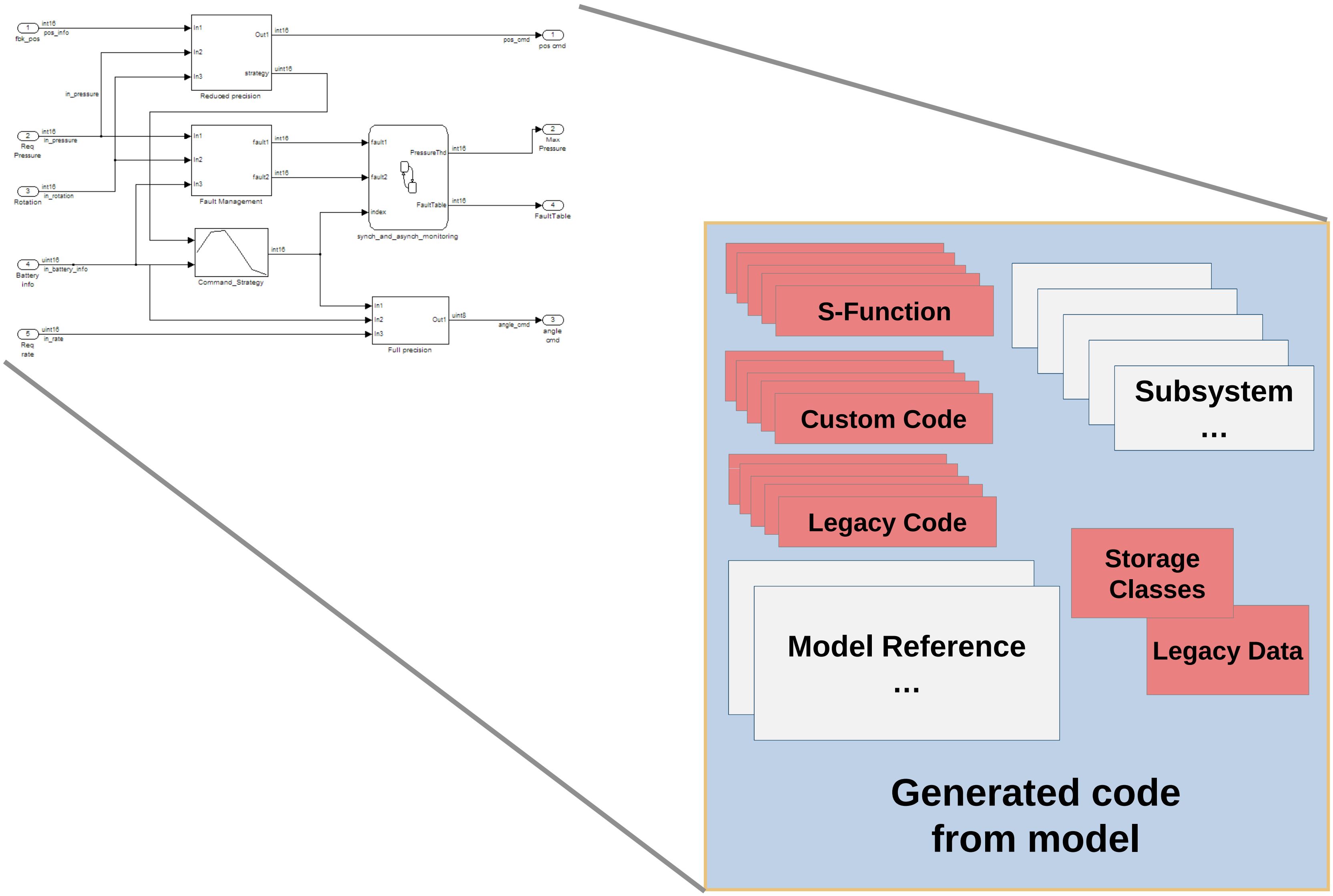
Model + Code

Aircraft Engine

Embedded Controller

Code

Automatic Code Generation from Model



S-Function

Custom Code

Legacy Code

Subsystem
…

Model Reference
…

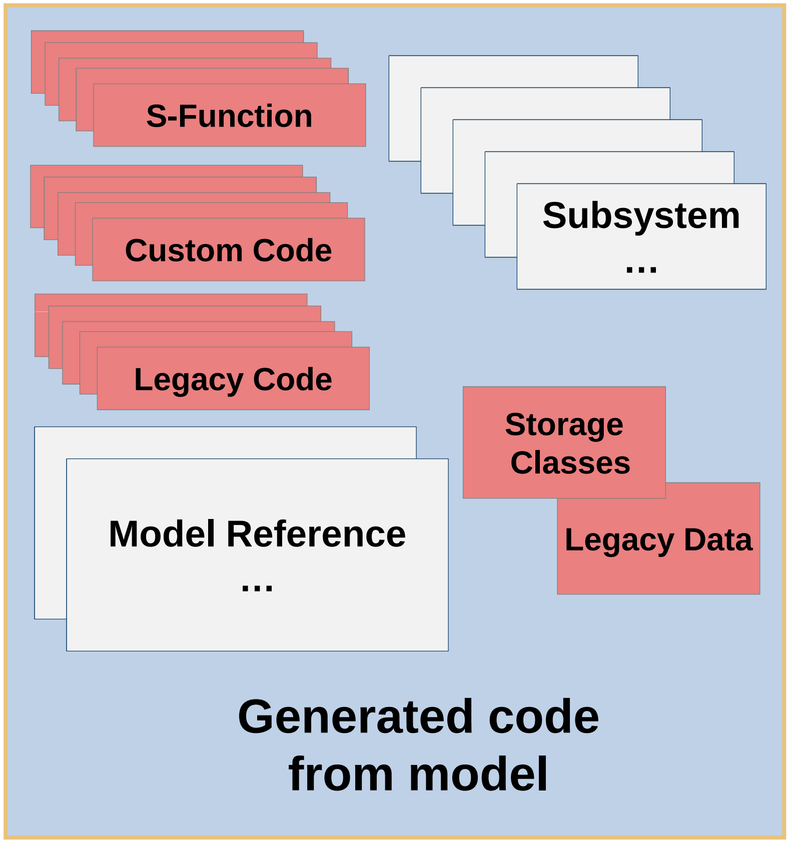
Storage Classes

Legacy Data

Generated code from model

Automatic Code Generation from Model

- Generated code consists of
  - Subsystems and model references
- Often includes handwritten code
  - S-Functions and legacy code
  - Individually, small in size (100s LOC)
  - May be automatically repeated many times within generated code

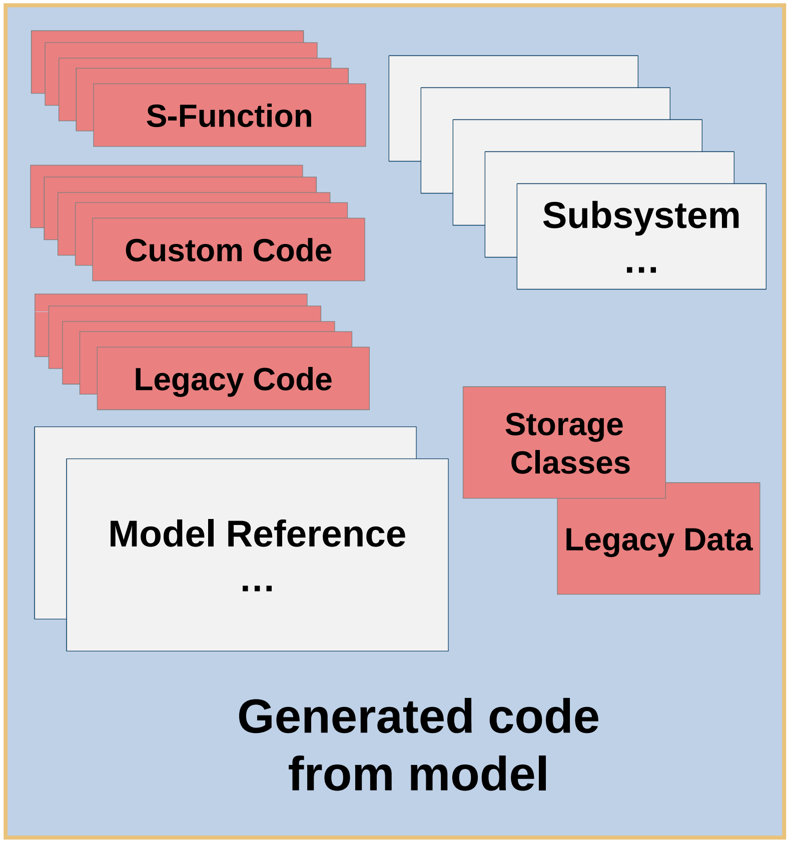


Automatic Code Generation from Model

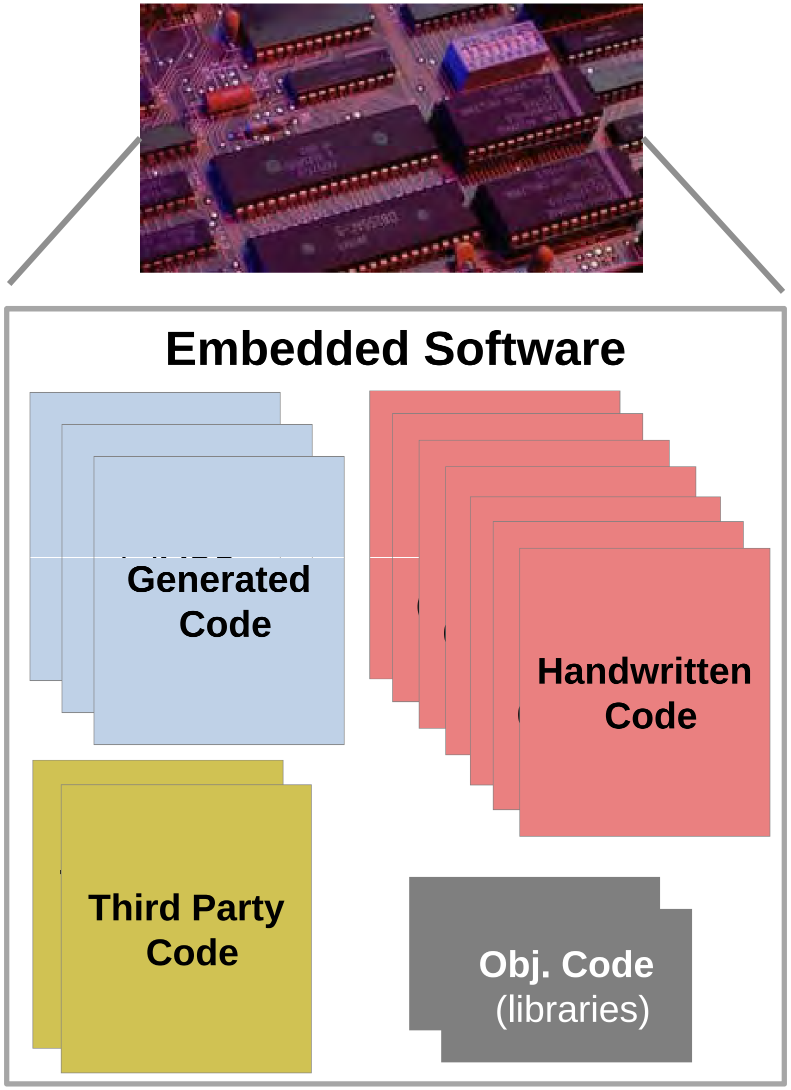
- Generated code consists of
  - Subsystems and model references

- Often includes handwritten code
  - S-Functions and legacy code
  - Individually, small in size (100s LOC)
  - May be automatically repeated many times within generated code

- Robustness issues to consider
  - Handwritten code fails, or causes generated code to fail
  - Generated code may cause handwritten code to fail (*Interface related failures*)
  - Handwritten code is not visible to modeling tools



Integration of Generated Code



Embedded Software

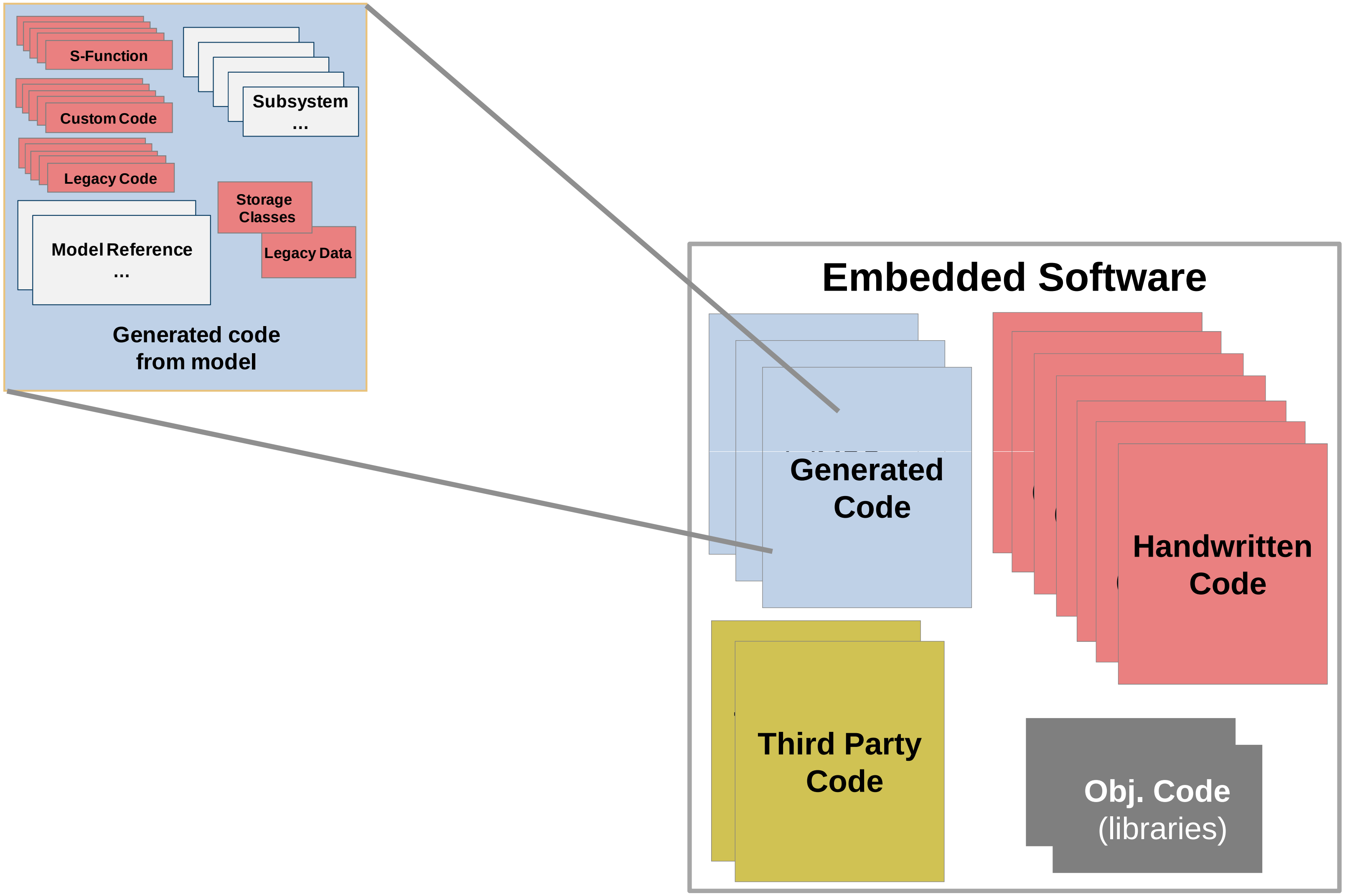
Generated Code

Handwritten Code

Third Party Code

Obj. Code (libraries)

Integration of Generated Code



S-Function

Custom Code

Legacy Code

Subsystem
…

Storage Classes

Legacy Data

Model Reference
…

Generated code
from model

Embedded Software

Generated Code

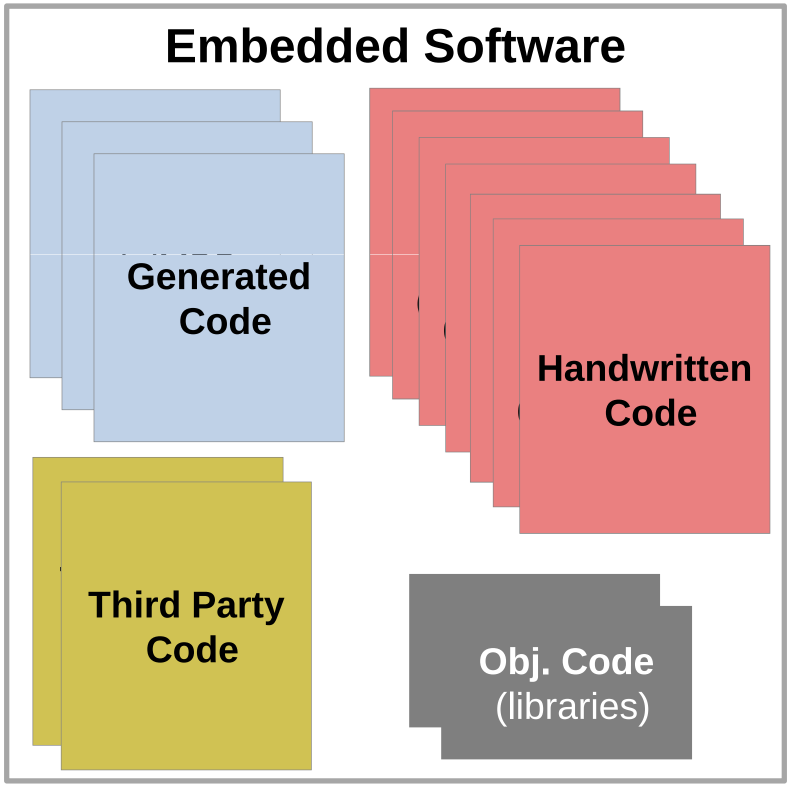
Handwritten Code

Third Party Code

Obj. Code
(libraries)

Integration of Generated Code

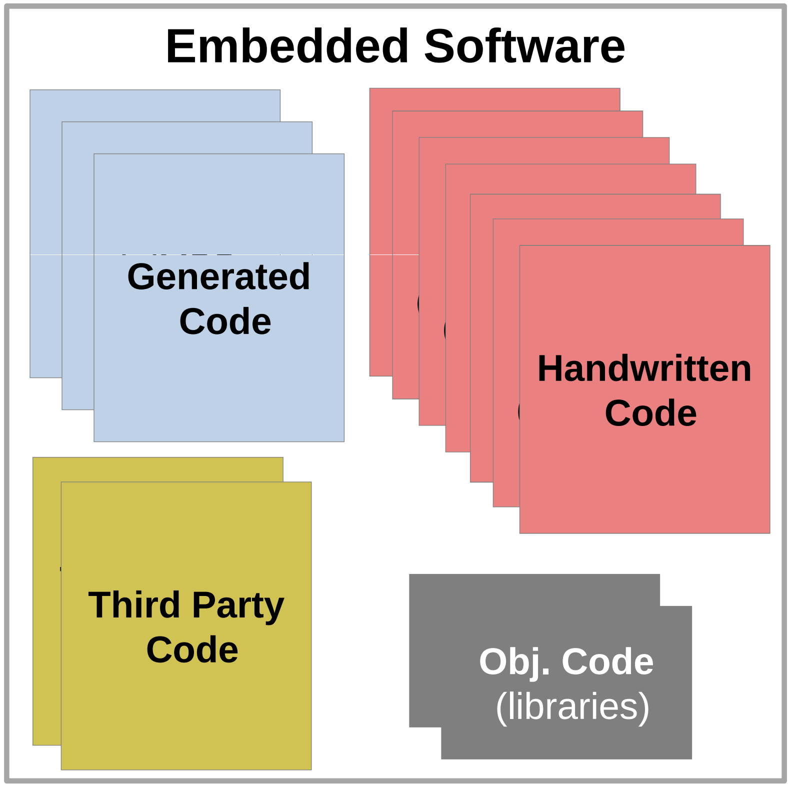
- Code integration
  - Generated code stitched together with handwritten code
  - All components integrated with handwritten code



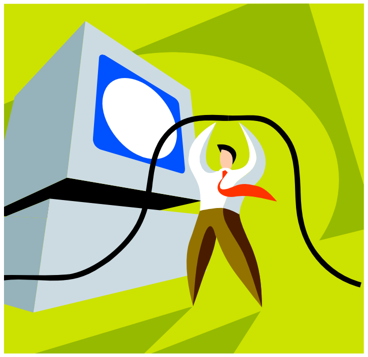
Integration of Generated Code

- Code integration
  - Generated code stitched together with handwritten code
  - All components integrated with handwritten code

- Robustness issues to consider
  - Design error in the generated code
  - Runtime error in handwritten or 3rd party code
  - How do you ensure the entire system is robust?
  - How to verify generated code at interface level?



Verification of Mixed Model and Code



Tutorial Demo

- Checking handwritten code in the models
- Verifying the generated code
- Verifying integrated code

Additional Techniques for Improving Software Quality

Getting near to zero defect goal

Enforce Code Standards

- C is a very flexible language
  - **`char **********ptr;`** is valid syntax
  - You can also write code without comments
- Are these good practices?
  - In general, NO
- Important to follow some code standards
  - Examples: MISRA C/C++, JSF++

Using Code Standards

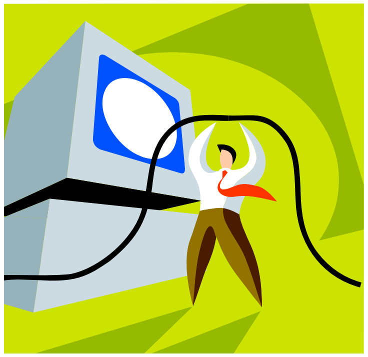
- Example standards
  - MISRA (Motor Industry Software Reliability Association), developed for automotive, but used outside in other industries
  - JSF++ (Joint Strike Fighter Air Vehicle C++)
- Facilitate
  - Code safety, portability and reliability
- Code rules
  - Some required, others advisories
  - Various categories, such as style, environment, and run-time

Example MISRA Rules

- Required
  - All object and function identifiers shall be declared before use
  - The right hand side of a *"&&"* or *"||"* operator shall not contain side effect
  - The statement forming the body of an *"if"*, *"else if"*, *"else"*, *"while"*, *"do ... while"*, or *"for"* statement shall always be enclosed in braces

- Advisory
  - Should not directly use basic types such as char, int, float etc.
  - All declarations at file scope should be static where possible
  - Tests of a value against zero should be made explicit, unless the operand is effectively Boolean

Applying Coding Standards



Tutorial Demo

- Application of MISRA C coding standards
- Measuring the improvement in quality

Enabling Software Quality

- Ideal goal, create software with zero defects

- In reality, must have a quality mandate
  - Internally or required externally
  - To meet specific software quality objectives

- Define a quality model with objectives
  - Enables a prescriptive solution to achieve quality

Runtime Defects in Software

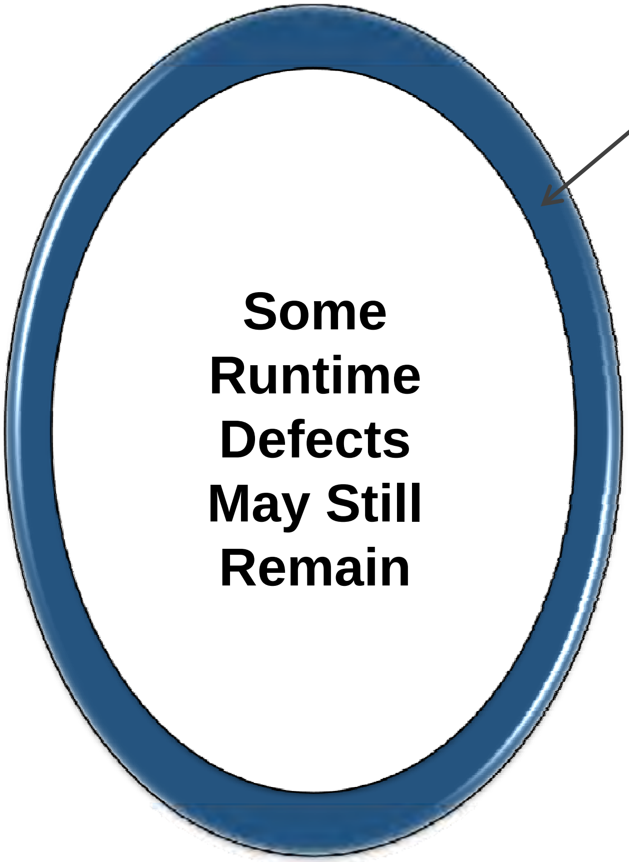
All Runtime Defects in Your Software

- Software will contain runtime defects
  - Cannot eliminate all defects in one step
- Incremental processes are needed
  - Different quality objectives and levels
- Ex. quality model with objectives
  - Six levels, s/w quality objectives (SQO)
  - For intermediate development and verification stages

Incremental Steps to Achieve Quality

All
Runtime
Defects
in Your
Software

Incremental Steps to Achieve Quality



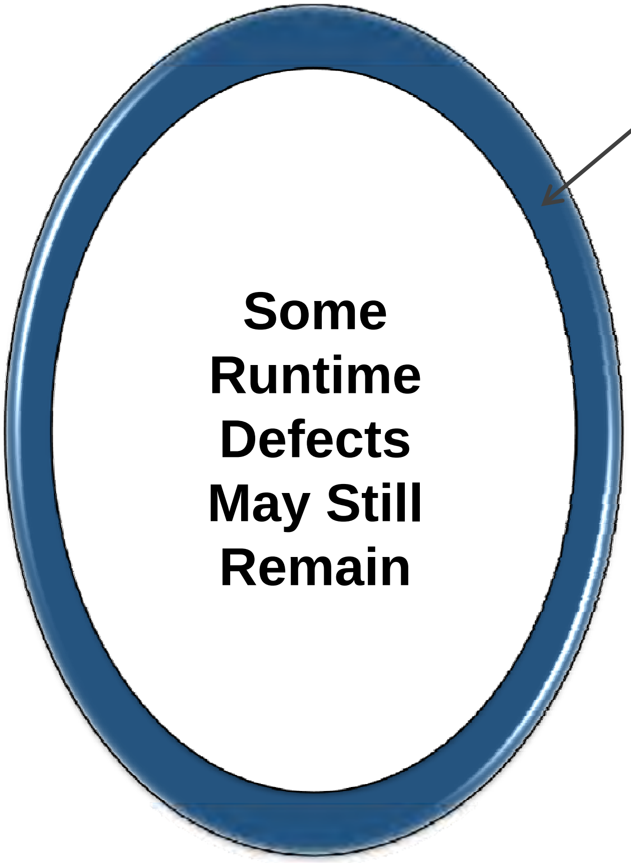
Some Runtime Defects May Still Remain

Eliminate some runtime defects

- By quantifying code verification results
  - Red, Gray, Orange
  - MISRA Rules
  - Code complexity metrics

Incremental Steps to Achieve Quality

Software Quality Objectives #1



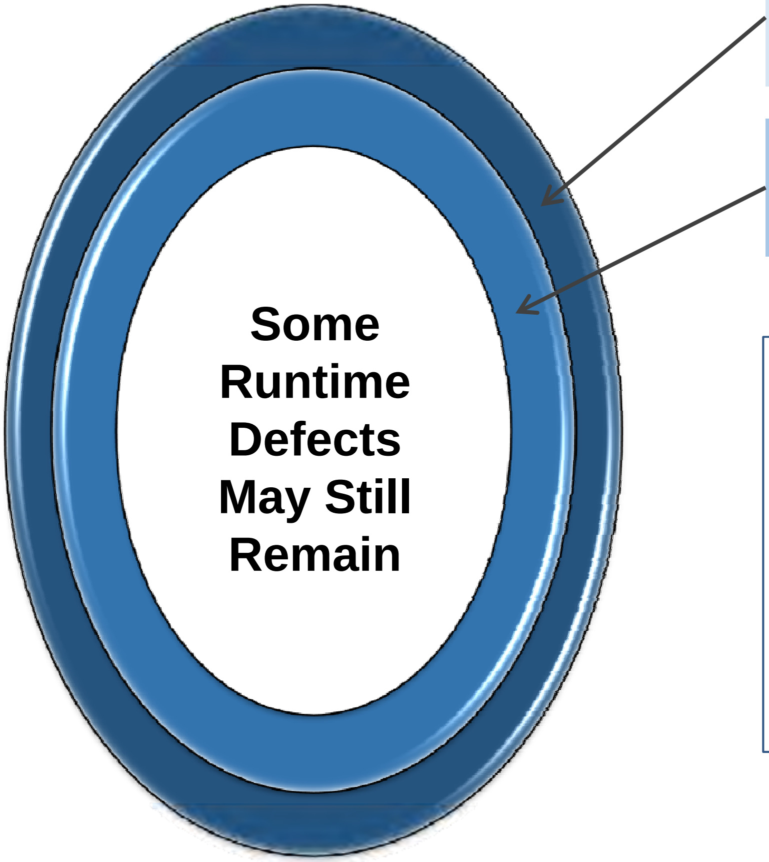
Some Runtime Defects May Still Remain

SQO1

- Meet specific code complexity thresholds
- Compliant to defined 1st MISRA-2004 rules subset

- First level has limited scope
  - Subsequent levels increase scope
  - Runtime defects may still remain in code

Incremental Steps to Achieve Quality



SQO1

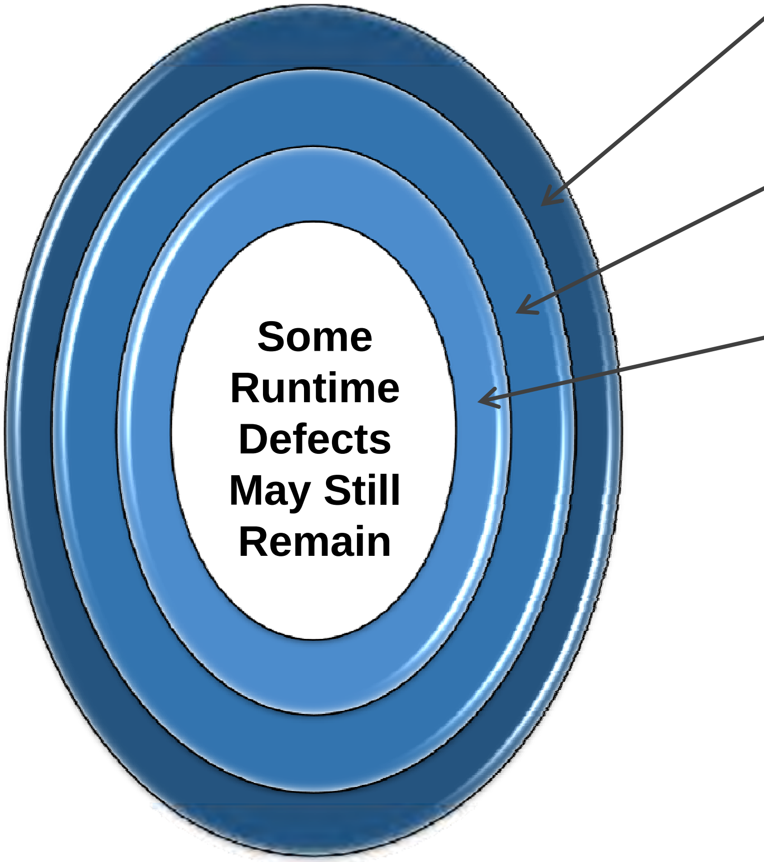
- Meet specific code complexity thresholds
- Compliant to defined 1st MISRA-2004 rules subset

SQO2

- No systematic run-time errors (i.e. no reds)
- No unintentional non-terminating constructs

- Second level increases scope
  - More runtime defects eliminated
  - But, runtime defects may still remain
- For an intermediate delivery
  - Subsequent levels will improve quality

Incremental Steps to Achieve Quality



Some Runtime Defects May Still Remain

SQO1

- Meet specific code complexity thresholds
- Compliant to defined 1st MISRA-2004 rules subset

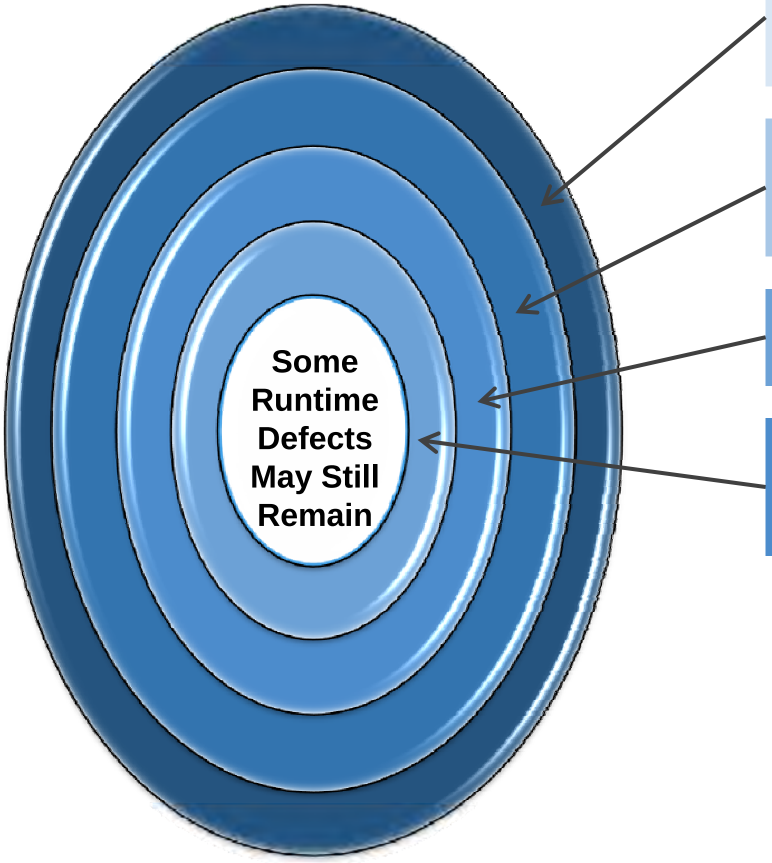
SQO2

- No systematic run-time errors (i.e. no reds)
- No unintentional non-terminating constructs

SQO3

- No unreachable branches (i.e. no dead code)

Incremental Steps to Achieve Quality



SQO1

- Meet specific code complexity thresholds
- Compliant to defined 1st MISRA-2004 rules subset

SQO2

- No systematic run-time errors (i.e. no reds)
- No unintentional non-terminating constructs

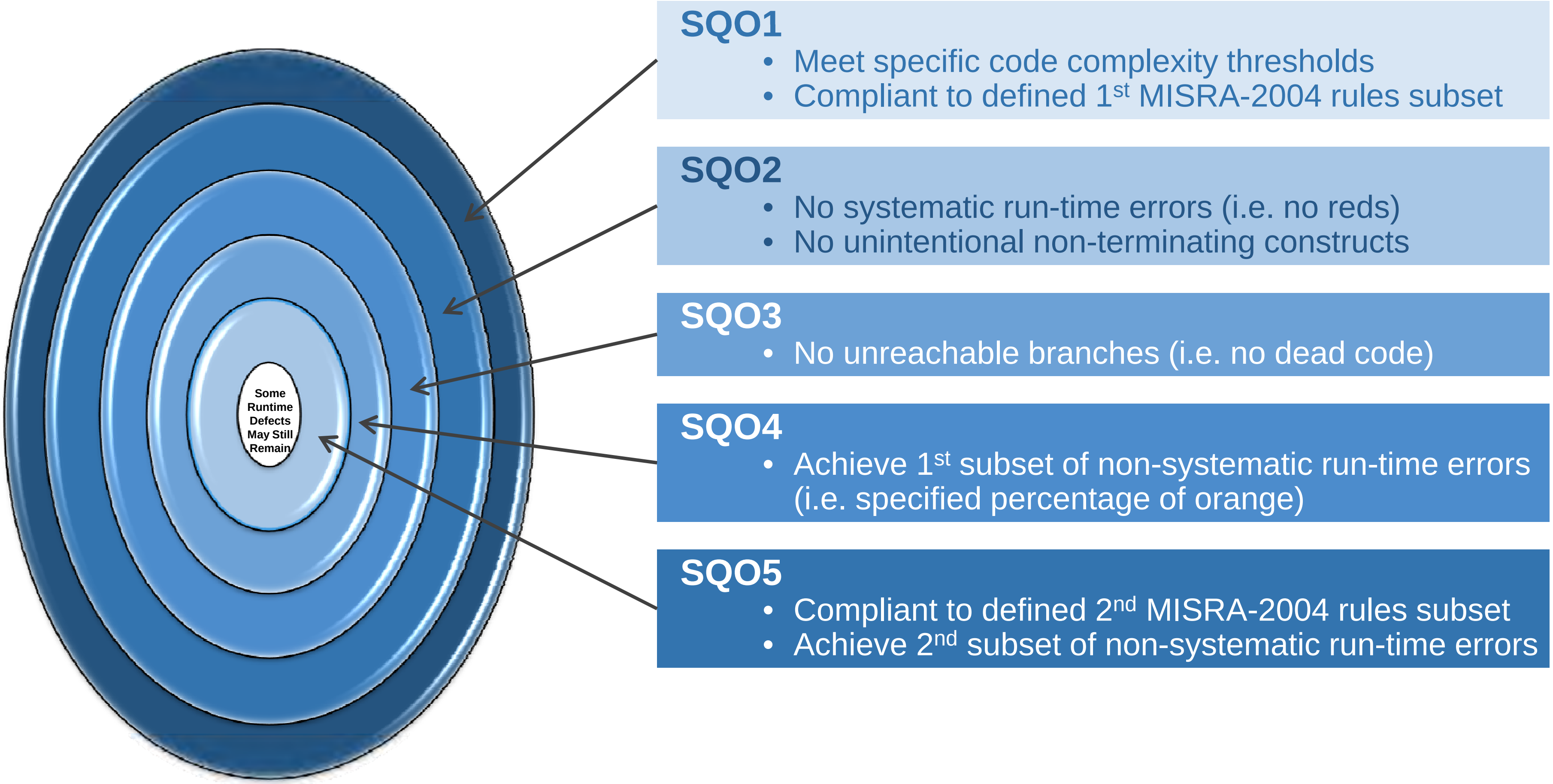
SQO3

- No unreachable branches (i.e. no dead code)

SQO4

- Achieve 1st subset of non-systematic run-time errors (i.e. specified percentage of orange)

Incremental Steps to Achieve Quality



SQO1

- Meet specific code complexity thresholds
- Compliant to defined 1st MISRA-2004 rules subset

SQO2

- No systematic run-time errors (i.e. no reds)
- No unintentional non-terminating constructs

SQO3

- No unreachable branches (i.e. no dead code)

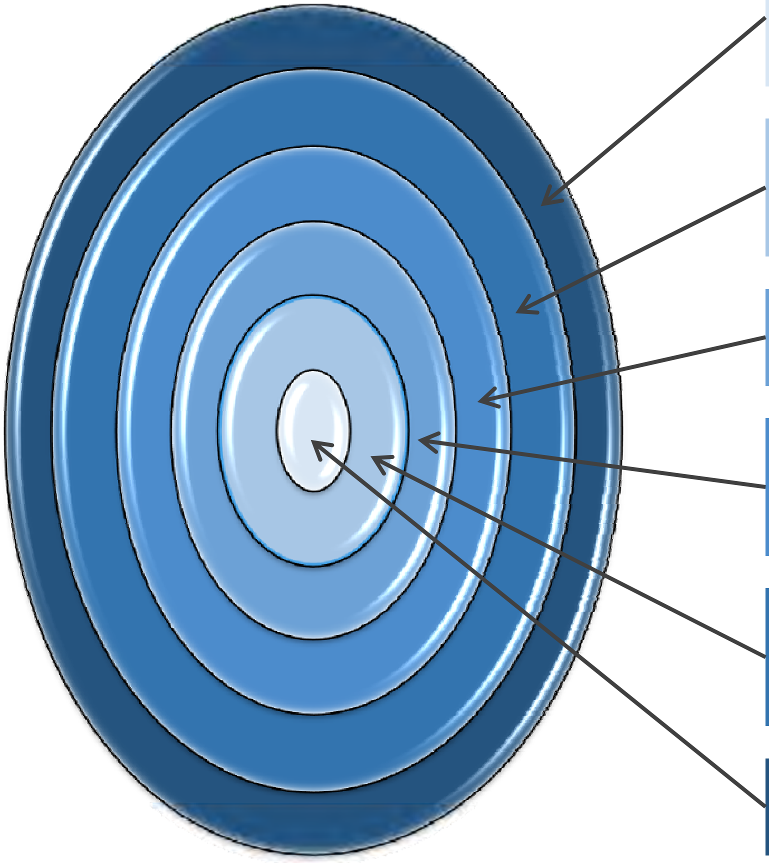
SQO4

- Achieve 1st subset of non-systematic run-time errors (i.e. specified percentage of orange)

SQO5

- Compliant to defined 2nd MISRA-2004 rules subset
- Achieve 2nd subset of non-systematic run-time errors

Incremental Steps to Achieve Quality



SQO1

- Meet specific code complexity thresholds
- Compliant to defined 1st MISRA-2004 rules subset

SQO2

- No systematic run-time errors (i.e. no reds)
- No unintentional non-terminating constructs

SQO3

- No unreachable branches (i.e. no dead code)

SQO4

- Achieve 1st subset of non-systematic run-time errors (i.e. specified percentage of orange)

SQO5

- Compliant to defined 2nd MISRA-2004 rules subset
- Achieve 2nd subset of non-systematic run-time errors

SQO6

- Achieve 3rd subset of non-systematic run-time errors

DO-178B Certification Credit with Verification Tools

- Partial credit for the following:
  - Table A-5
    - Ref. Section: 6.3.4b, 6.3.4c, 6.3.4d, 6.3.4f
  - Table A-6
    - Ref. Section: 6.4.2.1, 6.4.2.2, 6.4.3

- Next slide explain *6.3.4.b* and *6.3.4.f*

Certification Credit for 6.3.4.b

- Objective
  - Compliance with the software architecture
  - The objective is to ensure that the Source Code matches the data flow and control flow defined in the software architecture
- How tools can be used
  - The data flow
    - Prove adherence to this aspect of the standard, as it automatically builds global data dictionary and identification of shared data reading and writing accesses
- Artifacts
  - Data dictionary, concurrent access graph, etc.

Certification Credit 6.3.4.f

- Objective
  - Determine the consistency of the Source Code, including stack usage, fixed point arithmetic overflow and resolution, resource contention, worst-case execution timing, exception handling, use of uninitialized variables or constants, unused variables or constants, and data corruption due to task or interrupt conflicts
- Code verification helps to identify
  - Exhaustively: Fixed point arithmetic overflows, use of uninitialized variables and constants, etc.
  - Partially: Unused variables and constants
- Artifacts
  - Color coding to identify quality of code
  - Report generation for artifact purpose

Conclusion

Summary of tutorial

Adopting New Processes
Short Term

- Detect and fix design and code errors
  - Unreachable states, dead logic, etc.
  - Fix code level run-time errors

- Simplify code review process
  - Take verification results to code review

- Develop better test-cases
  - Improve coverage analysis
  - Understand impact of variable ranges

Adopting New Processes
Long Term

- Make verification a part of your quality improvement process
  - Monitor quality and status

- Leverage verification for certification
  - Maybe possible to skip some processes
  - E.g. show code does not contain divide by zeros

Conclusion

- Complexity of systems
  - Learn from past failures
- Model and code verification
  - Address design and code with error detection and proof
  - Use model verification to detect and fix design errors
  - Use code verification to detect and fix coding errors
- Practical considerations
  - Improve robustness in mixed model and code environments
- Additional techniques for improving software quality
  - Coding standards such as MISRA and JSF
  - Certification standards such as DO-178B
  - Achieving quality goals with software quality objectives

Acronyms

- DSP – Digital Signal Processor
- JSF – Joint Strike Fighter
- KLOC – Thousands (K) of Lines of Code
- LOC – Lines of Code
- MBD – Model Based Design
- MCU – Micro Control Unit
- MISRA – Motor Industry Software Reliability Association
- MLOC – Millions of Lines of Code
- SW – Software
- SQO – Software Quality Objectives